UNIVERSITY OF OSLO
Department of Physics

Implementation of a communication protocol for CubeSTAR

Master thesis

Markus Alexander Grønstad

July, 2010

Abstract

This thesis describes the process of implementing the software part of a communication sub-system for a student built satellite (CubeSat). The implementation consist of a communication protocol in the data link layer that utilizes AX.25 frames for communicating with a ground station based on amateur radio equipment. The development and usage of this communication protocol are based on data communication theory, link budget calculations, efficiency and availability analyzes together with practical tests. The steps in how this software has been developed and evaluated are described, and can be useful for other project that could benefit from this communication protocol. Source code listings are included. The reader should be familiar with communication theory, data communication, programming and electronics.

Acknowledgement

Some fantastic years at the University have come to an end, and a long journey as an academic ambassador has started. In my B.Sc. and M.Sc. degree I have explored and studied the fundamentals of technology that always have fascinated me.

My masters project started one year ago at the Department of Physics. The opportunity to combine several interesting fields as digital communication, programming and electronics made this project really interesting for me. It has been highly enjoyable to see the results from practical implementations of theory in this project, and having the final project goal of a functional satellite.

I would like to thank associate professor Torfinn Lindem for the opportunity to take this masters project. A great thanks also to the members of the CubeSTAR project, and especially Johan Tresvig, Eirik Vikan, Henning Vangli and also Mats Randgaard for sharing knowledge, humor and offices the last year.

The study life at the University would not have been the same without all the social influence by Realistforeningen, thanks to H.M. Ursus Major for this.

A great thanks to my parents for all the needed support provided the last 26 years, and also for keeping me structured and focused. It is highly appreciated.

Finally my sincere thanks to Sunniva for always being the sweetest girl. Together with our daughter Alexandra, you bring more joy and motivation to my life than I could ask for.

Markus Grønstad
Oslo, July 2010

Contents

Acronyms

The following acronyms are used in this thesis.

ACK Acknowledged message

ADCS Atitude Determination and Control System

AFSK Audio Frequency Shift Keying

AOS Acquisition of Signal

APRS Automatic Packet Reporting System

ARQ Automatic Repeat Request

ARS Andøya Rocket Range

BER Bit Error Rate

CCSDS Consultative Committee for Space Data Systems

CNES Centre National d'Etudes Spatiales

COMM Communication

CRC Cyclic Redundancy Check

CW Continuous Wave

Cal Poly California Polytechnic State University

EIRP Equivalent Isotropically Radiated Power

EPS Electrical and Power System

FCS Frame-Check Sequence

FEC Forward Error Correction

FSK Frequency Shift Keying

GENSO Global Educational Network for Satellite Operations

GEO Geostationary Earth Orbit

GFSK Gaussian Frequency Shift Keying

HAL Hardware Abstraction Layer

HDLC High-Level Data Link Control

HiN Narvik University College (Høyskolen i Narvik)

IARU The International Amateur Radio Union

ITU International Telecommunications Union

LEO Low Earth Orbit

LNB Low Noise Block

LOS Loss of Signal

LSB Least Significant Bit

MEO Medium Earth Orbit

MET Maximum Elevation at Transit

MSB Most Significant Bit

NACK Not-Acknowledged message

NTNU Norwegian University of Science and Technology (Norges Teknisk, Naturvitenskapelige Universitet)

OBDH On-Board Data Handling

OOK On-Off Keying

OSI Open Systems Interconnection

PCM Pulse-Code Modulation

PER Packet Error Rate

PID Protocol Identifier

RF Radio Frequency

SNR Signal-Noise-Ratio

SSB Single Side-Band

SSID Secondary Station Identifier

TLE Two Line Element

TNC Terminal Node Controller

UI Unnumbered Information

UiO University of Oslo (Universitetet i Oslo)

List of Figures

List of Tables

Chapter 1

Introduction

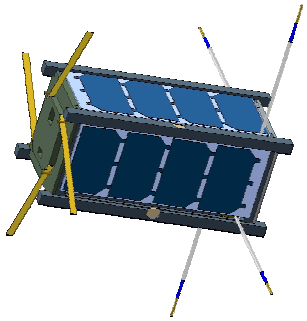


Figure 1.1: The CubeSTAR satellite.

This thesis is submitted for my masters degree in Electronics and Computer Technology at the University of Oslo (UiO), Department of Physics. The masters project has been a part of the CubeSTAR[1] project.

The CubeSTAR is the project name of a student satellite that is to be built at UiO, according to the CubeSat standard. This standard is written by California Polytechnic State University (Cal Poly) and Stanford University, and defines a cubic satellite structure with dimensions 10 cm x 10 cm x 10 cm and a weight of 1 Kg. These structures can be assembled into 1, 2 or 3 units. The standard defines a pod for storing CubeSats during launch, this makes it possible for several CubeSats to piggyback on a launch of a greater satellite. Thus keeping the costs at a level that can be affordable for academic institutions.

Several universities around the globe have CubeSat projects going, and have launched one or several CubeSats. Different design strategies are used, varying from the assembling of commercially available CubeSats sub-systems, to

[1]The name "CubeSTAR" comes from the concatenation of Cube from CubeSat, and STAR from the Space Technology And Research center at the UiO.

the design and implementation of the CubeSat at component level.

In the CubeSTAR project, a goal is to build and develop most of the systems in house. This introduces several interesting masters projects that together will have a final goal. Most of the satellite systems will be built and developed by students at the University facilities. The electronics laboratory and the instrumental workshop at the Department of Physics will provide the satellite structure and backplane bus.

A scientific experiment will be the payload of the satellite. This payload is an instrument developed by Tore André Bekkeng[1], based on an idea and concept by Knut Stanley Jacobsen[2]. In the experiment, Langmuir probes are used to measure electronic density in the F layer of the ionosphere.

The satellite will consist of several sub-systems, ranging from Electrical and Power System (EPS), Atitude Determination and Control System (ADCS), On-Board Data Handling (OBDH) and Communication (COMM). The CubeSTAR satellite will be a 2 unit structure, measuring 20 cm in length and 10 cm x 10 cm in base, with measuring probes and antennas standing out. A modell of the satellite can be seen in Figure 1.1 on the preceding page. The launch of the satellite is scheduled to 2012. At this written time the orbital parameters and orbit height is not yet available, although an orbit height of 300–400 km is expected. The calculations in this master thesis are based on these expectations wherever it is necessary.

In my masters project the motivation was to find, adapt and test a suitable communication protocol that could be compatible with other CubeSat missions and the Oslo Ground Station. This ground station have been set up at the University facilities, according to the Global Educational Network for Satellite Operations (GENSO) project. GENSO aims to connect several ground stations together to a distributed network, where ground station resources can be shared among CubeSat projects.

My implementation of the communication protocol will need to be interoperable with the Oslo Ground Station, and standards defined by GENSO. It will have to support the hardware chosen for the satellite side communication sub-system. To determine how effective the protocol can be, I wanted to make an evaluation so the CubeSTAR project could have measurements for what to expect in therms of telemetry throughput and how the communication protocol can be used efficiently.

Some of the challenges in this project was to find good references from others work on their communication software for CubeSat missions. Although Norway has had two previous CubeSat missions (Ncube I & II), unfortunately both missions failed[2]. Documentation from these projects that could illumi-

[2] NCUBE-1 did not make it into orbit, as the DNEPR-1 vehicle failed short time after launch, destroying the 18 satellites carried on-board. NCUBE-2 was supposedly delivered

nate some of the issues encountered in my project has not been available. The previous CubeSat missions in Norway have been projects at the Norwegian University of Science and Technology (NTNU) and Narvik University College (HiN). HiN have a new CubeSat project going, called HiNCube. Hence the CubeSTAR is the fourth Norwegian student built CubeSat, and the first one at the UiO.

I chose to implement a version of the AX.25 protocol, this is an extension of the X.25 protocol for transferring packet data over amateur radio network. The implementation consists of a communication software for the microcontroller in the communication sub-system, that can communicate with a ground station over a communication protocol based on AX.25 frames. Commercially available amateur radio equipment supports AX.25, this way the communication system on the ground station side is not dependant on custom hardware implementations. Thus making communication with CubeSTAR more available since we already have an existing ground segment.

The practical use and implementation of AX.25 has proven to be challenging due to the documentation. Most of the physical layer is more or less based on *de facto* standards in the amateur radio community. This thesis will describe how to implement a communication protocol based on AX.25 framing, where the physical layer standards are explained. This will not only be valuable for the CubeSTAR project, but hopefully also for other CubeSat missions. The source code consists of a library for encoding and decoding AX.25 frames that easily can be adapted for other microcontrollers, hardware and communication architecture. The evaluation of the implementation is based on practical tests, efficiency calculations and what constraints that applies in the link analyzes.

All source code developed in this project are to be found in the enclosed CD-ROM at the back cover of the printed version of this thesis. It is also available upon email request to markusg@ieee.org.

Chapter 2 describes digital communication fundamentals and theory for how to transfer digital data over a Radio Frequency (RF) wireless channel, and what higher abstraction layers are needed to manage a reliable communication. In the end of this chapter I have defined the approaches for my implementation.

Chapter 3 presents satellite communication fundamentals and a link analyzis of a communication link between a satellite and a ground station. This will illustrate what limitations and margins that affect the rest of the communication architecture, based on the data rates available for the communication protocol defined in the previous chapter.

into orbit, but no contact have been achieved.

Chapter 4 continues the communication theory from the first chapters by describing the data link layer, and how to implement the chosen protocol for CubeSTAR. What services this implementation would serve to other communication layers are described together with how theory from Chapter 2 can be implemented.

Chapter 5 describes the implementation of the communication protocol, and what software and hardware that was used. The program flow for how the communication sub-system utilizes the communication protocol are described, together with how it can be integrated with the OBDH and other firmware in the communication sub-system. The process of verifying correct encoding and transmission, and correct reception and decoding are described.

Chapter 6 uses results from the implementation together with the communication theory to evaluate the efficiency of the communication protocol in a communication architecture. Different strategies for utilizing the implementation are discussed and analyzed. How the implementation meets the preliminary mission specific requirements are calculated and presented. These numbers can serve as guidelines for what to expect in telemetry throughput from the CubeSTAR mission.

Chapter 7 presents the test methodologies of the communication protocol and the test applications developed for testing the implementation. How these applications are used and the results obtained are described in this chapter.

Chapter 8 elaborates the implementation through a final evaluation, other alternatives, and the future work remaining for integrating the proposed implementation to the rest of the systems in CubeSTAR. The final conclusion will sum up what presented in this introduction and described in the thesis.

Chapter 2

Digital communication

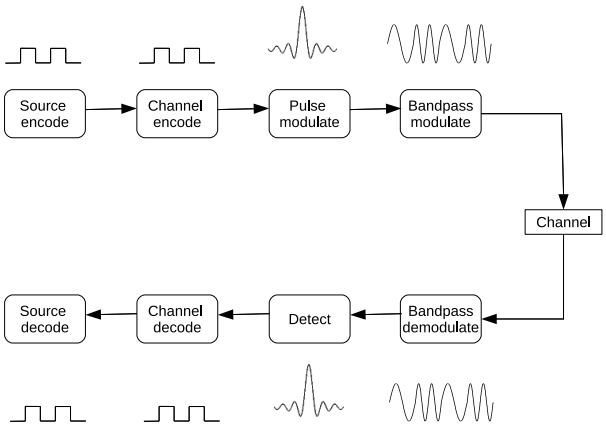


Figure 2.1: A digital communication system overview.

The communication system overview displayed in Figure 2.1 shows the building blocks of a digital communication system, together with an illustration of in what format the signals and waveforms are represented throughout the system. The combination of these individual blocks describes the process in converting digital data to a RF signal, transfer it over a channel, and recovering the sent data at the receiver.

These fundamental parts of the communication system overview are ex-

plained relative to how they can be implemented in the CubeSTAR satellite.

Source encoding

Source encoding is the process of producing an efficient digital representation of the data to be transmitted. This data can either be collected from analog or digital sources. Several processes can be applied to the sources for building up the final informational data, ready for transmission. These processes range from sampling and quantizing of analog data, to different coding techniques of the digital data. Compression can be applied to reduce the size of the data. The purpose of source encoding is to reduce the need for system resources of bandwidth and/or energy per bit by removing some of the redundant data.[3]

In the CubeSTAR satellite, the source encoding process can be distributed to other systems outside the communication sub-system. The payload and OBDH systems can responsible for the source encoding. Housekeeping data from the satellite and the telemetry data from the payload could be structured into a format, before delivered to the COMM system.

Channel encoding

Where redundancy was removed from the informational data in the source encoding process, extra redundancy is now added again. This way it is possible in some degree to reconstruct the information carried in the transmitted signal, even if it is corrupted by noise, interference or fading.

Different techniques exists for error control and these are often incorporated in the communication protocols, Section 2.1.1 describes common error handling techniques that will be available for CubeSTAR.

Pulse modulation

Before pulse modulation, all the data is in a digital discrete format. This data now needs to be represented as electrical pulses before bandpass modulation. The bit stream will be converted to a pulse train where a binary one is represented as a pulse, and a binary zero represented as the absence of a pulse. When a binary symbol is pulse modulated, the resulting waveform is a Pulse-Code Modulation (PCM) waveform, also called for line coding. There are different PCM waveforms, with different characteristics. [3, p. 85] The filtering process in the pulse modulating process have effect on the bandpass modulation,

Bandpass modulation

From the pulse modulation, the signal needs to be modulated for the channel. This process involves mixing the shaped pulses with a sinusoidal carrier frequency. To represent the binary symbols in bandpass, the carrier frequency can change the characteristics of the signal in either amplitude, frequency, phase or a combination of them. The collection of these techniques are called modulation schemes.

The pulse modulation together with the bandpass modulation will in the CubeSTAR satellite be performed in the radio interface of the transceiver circuit.

Channel

The channel is the medium that the bandpass signal propagate. In a wireless communication system as a satellite link, the channel is the open air. This is where the primary source of error exists. The transmitted signal attenuates due to the path loss in the distance between the transmitter and receiver. In a satellite link there could be atmospheric conditions that causes additional attenuation to the signal. Other signals in the channel are treated as noise, and could interfere with the transmitted signal. The relationship between the informational signal and noise, is expressed as the Signal-Noise-Ratio (SNR). A signal with low SNR could cause errors in transmission. Hence data delivered with errors.

$$C = W \log_2 \left(1 + \frac{S}{N}\right) \tag{2.1}$$

By the Shannon-Hartley theorem, Equation 2.1, the capacity of the channel is shown. This defines the theoretical limit given in bits/s that it is possible to transmit at a given SNR, and bandwidth W in Hertz. Information must be transmitted at a rate lower than this limit, to achieve a small enough error probability. Complicated coding schemes can utilize the channel near the limits of the Shannon-Hartley theorem,

Data rates

The data rate is measured in baseband and is often expressed as bits per second. In bandpass modulation it is possible to combine several bits into a symbol for higher throughput through the channel. Some modulation schemes can transfer several bits at a time. The modulation scheme then defines the relationship between bit rate and symbol rate. Baud rate is often

7.	Application
6.	Presentation
5.	Session
4.	Transport
3.	Network
2.	Data link
1.	Physical

Figure 2.2: The OSI reference model defines the abstraction layers in a data communication system.

used as an expression for the symbol rate. This can be confusing if bits per symbol is not defined. Bit rate is thereby used throughout this thesis to define the data rate in baseband.

2.1 Data communication

The previous section described the components of a communication system. But in order to manage the flow of data we need to define the higher abstraction layers for a reliable data transfer to exist.

A data communication system is often expressed as layers in a model referred to the OSI model. See Figure 2.2. This is a theoretic model, but describes the ideas in data communication well. The TCP/IP model is also a layered model that is derived from the OSI model. The principle of the layers is that a layer provides services to the above layers, and receives services from the layers below.

The services referred to in the layered model could be error detection and correction, management of multiple users, network routing, data presentation and so on. Different protocols are used to achieve this, but in order to ensure a reliable data transfer, we will need to present the unreliable raw data stream from the physical layer in a format that can be provided to the higher abstraction layers. This is where framing is used. The raw bit stream is divided into chunks of data, called frames. These frames can consist of the informational data together with addresses of sender and receiver, redundancy for error control and other protocol data. [4][5]

Defining the services that CubeSTAR will need to offer, and the overall communication architecture, will form the basis of how the communication protocol will need to be structured.

2.1.1 Error handling

Error handling will need to be introduced early in the communication subsystem to ensure reliable data transfer. The redundancy added in the channel encoding process can be used for detecting errors, and correcting errors. This redundancy is called for parity bits, from the easiest way of detecting errors. A single bit can be used to mark if the frame has an even or odd number of 1-bits. This technique serves more as a theoretic example since it will not work for data frames containing multiple bit errors.

What technique that should be used for error handling should be defined by the probability for bit errors and the importance of correct reception of all the transmitted data. However the overall communication architecture and the error handling features that already are incorporated in the communication protocols could restrict the available techniques.

The terminal connectivity defines how the terminals or network nodes are connected together. Some of the error handling features relies on that transmission can occur in both directions, and other are able to ensure reliable communication in connectivity where transmission only can occur in one direction.

In a *simplex* connection, there is only transmission in one direction, e.g. from the satellite to the ground station. A *half-duplex* transmission can carry transmission in both directions, but not simultaneously. The *full-duplex* connectivity can carry transmission in both directions.[3]

Cyclic Redundancy Check

The CRC checksum is the result of a mathematical function that is calculated from the data to be transmitted, and applied to the data frame before transmitting. In the receiver the same function is used to calculate the checksum of the received data, the two checksums are matched and any errors in the received data will cause a mismatch, thus errors are detected. Like with a simple parity bit, it is not possible to detect the amount or distribution of bit errors in the data frame. Hence it is not possible to correct the erroneous data by only inspecting the checksums, and CRC is a error detecting technique unable to correct for detected bit-errors.

Forward Error Correction

Forward Error Correction (FEC) is a family of coding schemes that works without feedback from the receiver. The receiver can utilize the extra redundancy added, to calculate where possible bit errors have occurred, and

thereby correct them. The coding schemes are divided into block codes, convolution codes and turbo codes. These groups vary in what techniques are used for encoding and decoding, the amount of redundancy added and the complexity of the code. FEC can be very effective and some codes have performance near the theoretical limits of the Shannon-Hartley theorem. One of the advantages of FEC is that it can deliver error-free data in a one-way transmission link, as broadcasting. Together with the fact that transmission delays caused by encoding and decoding can be kept constant, makes FEC suitable for real-time services such as speech or video. However if bits are lost, or a code block can not be correctly decoded by the receiver, that given frame will in either hand be lost.

[6]

Automatic Repeat Request

Automatic Repeat Request (ARQ) is based on the principle of detecting transmission errors, and request the sender for retransmission of the erroneous data frames. The receiver will need to acknowledge for which data frames that was received correctly. Contrary to FEC, the transmitter will need feedback from the receiver in ARQ, frames sent will either need an Acknowledged message (ACK) or Not-Acknowledged message (NACK) from the receiver.

Often a CRC is used to detect if errors exists in the received data, together with a sequence numbering method in the frames for the detection of lost frames. It is also possible to combine the ARQ with a FEC. Thus making the communication system try correcting erroneous data before requesting for a retransmission.

By using ARQ it is possible to obtain an error free communication, and it is often used in data transmission where errors are not accepted, as in file transmissions.

Stop-And-Wait protocol is the simplest way of implementing ARQ. Each frame will either need an ACK or a NACK from the receiver. If the transmitter does not receive this message in a given time interval it must retransmit the last frame. This method is easy to implement, but gives a poor channel utilization and low data throughput as each frame will need to be acknowledged for, with the delays that this introduces.

Go-Back-N protocol is a technique that transmits a number of frames in blocks. The transmitter will then wait for acknowledgements from the receiver for all frames. If a NACK or a missing ACK is detected for one of the frames, the transmitter will resend all the frames in the block from the not acknowledged frame.

Selective-Repeat protocol uses a block transmission of frames. This is the most effective technique, but also the most complex to implement. As opposed to Go-Back-N, when a frame is not acknowledged for, Selective-Repeat only retransmits that given frame.

[6]

2.2 CubeSTAR communication

A reliable communication protocol for CubeSTAR will need to be based on the theory mentioned above, adapted for the hardware already for the project.

Since a satellite link is to be used, the communication protocol will need to be implemented for and taking account for this scenario.

The system trade-offs and constraints will need to be found and evaluated to find out that the communication protocol can meet the expected requirements.

2.2.1 Licensing and frequency bands

Choosing the frequency bands to operate the satellite in will except from the technical aspects have some legal aspects. Different licences are attached to different frequency bands. Using the frequency bands for the amateur radio network have been used by several CubeSat missions. The frequency bands available for amateur satellite operations are 144 – 148 MHZ, 435 – 438 MHz, 1260 – 1270 MHz and 2400 – 2450 MHz, there are also some higher frequencies available. It is important to note that other services are sharing these frequency bands.

The advantage of using the amateur radio bands, is that they can be used without formal notification and application to the International Telecommunications Union (ITU).

The International Amateur Radio Union (IARU) provides guidelines for how to fit in the criterion for using the frequency bands for amateur radio. The stations operating at the amateur frequency bands must be controlled by a licensed amateur radio operator. The international communication must be in plain language, meaning that the protocol and framing format must be public available. Commands for critical spacecraft functions can be excluded. All other communication should be open for use by amateur radio operators worldwide.

The purposes of an amateur satellite is defined by IARU as:

1. *"Provide communication resources for the general amateur radio community and/or"*
2. *"Self training and technical investigations relating to radio technique."*

If the mission plans are consistent with the guidelines provided by IARU, the frequency planning process should be continued in collaboration with the national amateur radio society.[7]

2.2.2 Packet radio

While discussing the usage of amateur radio frequencies, we need to look on how to meet the criterion in the communication sub-system of CubeSTAR. The ground segment of the CubeSTAR project, Oslo Ground Station is already equipped with amateur radio equipment. The amateur radio community have methods and standards for transferring packet data over radio and over satellite links. The most common standard is referred to AX.25, and various implementations are in use. Automatic Packet Reporting System (APRS) is a common implementation that uses the AX.25 protocol and framing format for sharing weather data, positional data and so on, among radio amateurs.[8] The data rates for AX.25 are usually either 1200 bps or 9600 bps. [9]

Various implementations of AX.25 have been used in other CubeSat projects.

2.2.3 Telemetry, Tracking and Command

The communication sub-system of a CubeSat together with the ground station is responsible for the Telemetry Tracking and Command (TT&C). By telemetry, measurements are made by the distance satellite, and transmitted to the ground station. The ground station can collect data from the satellite and/or from other sources for tracking the moving path of the satellite. Control of the satellite is established through commands from a ground station. [10]

2.2.4 The GENSO network

The GENSO project aims to connect several satellite ground stations around the world to a network. The motivation comes from the grow in interest of CubeSat mission. Although almost all CubeSat projects have different goals and satellite technology, the ground stations follow much of the same setup. They are mostly based on commercially available amateur radio equipment. Some of the main challenges all CubeSat missions encounter, is the short time of availability as the satellite is in line of sight of the ground station,

due to the Low Earth Orbit (LEO). Since many of the CubeSTAR projects are geographically distributed around the globe, the idea of sharing capacity and usage of ground stations between CubeSat projects would gain the data throughput to and from the satellite.

The Oslo Ground Station are set up according to hardware specified to be compatible with the GENSO project, and the communication protocol implemented in this project should be interoperable with GENSO.

At this written time, the GENSO project has not yet released public available software or documentation. A list of compatible hardware and support for the AX.25 protocol has been announced. [11]

2.3 Approaches

My approach was to find a communication protocol that could be implemented in the given time frame for a masters project. The communication protocol would have to support the hardware already on place at the Oslo Ground Station, and the hardware chosen for the communication sub-system on CubeSTAR. It must support a reliable service of data transfer for a two-way communication with the satellite. Together with the implementation I wanted to make an analyze of how effective the protocol will be in therms of delivering telemetry data.

I decided to implement a version of AX.25 since it seemed to be a well proven protocol from other CubeSat missions, and the fact that it is supported by off the shelf amateur radio equipment. This would make it interoperable with already existing equipment at the ground station side. The hardware that has been chosen for the communication sub-system on the satellite side, has to be able to communicate by AX.25 frames with the ground station hardware.

Before starting with the implementation I wanted to understand the satellite orbits for CubeSTAR and what requirements an AX.25 implementation will set on the rest of the communication system. Since the project was at an early stage when my masters project was given, the rest of the satellite systems was not started and no functional nor performance requirements for the communication protocol was set. I decided to provide the implementation as a framework with a library for encoding and decoding AX.25 frames, and a software for the microcontroller in the communication sub-system that uses this library for communication.

I wanted to provide a test framework for the communication protocol to ensure the functional usage and evaluate the performance. This framework should be easy to use for debugging and evaluating the protocol. It should

be adaptable for other tests when the communication protocol is to be integrated with the rest of the system. By using a graphical user interface for the test setup, this process will be easier for other to perform.

Analyzing the performance of the communication protocol would give the CubeSTAR project results in how the requirements for the scientific projects are met and what constraints and trade-offs that must be evaluated.

Chapter 3

Satellite Communication

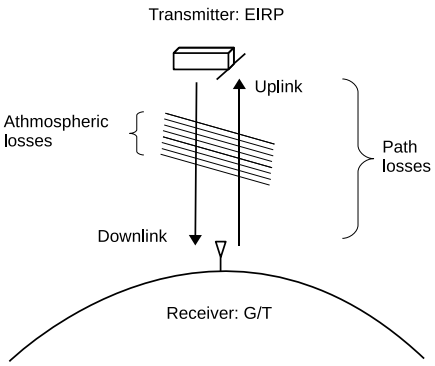


Figure 3.1: Satellite communication.

In a space system the satellite and ground station must be in line of sight of each other, and using frequencies above 100 MHz for penetrating the Earths ionosphere, to achieve a communication link.[12]

3.1 Satellite orbits

Understanding the satellites orbit is important in analyzing the communication link. We will need to describe where the satellite is relative to the earth.

Satellite orbits can be divided into these three general orbit types:

Geostationary Earth Orbit (GEO), has an altitude of 35 786 km. With this fixed height and an inclination angle of 0° it will appear stationary. The inclination is defined as the angle where the orbit crosses the equatorial plane. This will give the satellite 24 hour period, and observed from the earth it will have a fixed position above equator.

Medium Earth Orbit (MEO), has altitudes between 5000–10 000 km.

LEO, has altitudes between 500–1 500 km. This is the most interesting orbit type for CubeSats, due to low altitudes. The small size of a CubeSat restricts the practical usage in higher orbits, e.g. high enough power for the communication system will be difficult to achieve. These altitudes are also wanted for the scientific experiment of CubueSTAR.

Given the parameters from the orbital plane in Figure 3.2 on the facing page, we can see that when the semimajor and semiminor axes are equal length, i.e. $a = b$, the orbit is circular.

The satellite orbit of CubeSTAR will be a circular LEO, thus the following distance and elevation calculations are based on that orbit type.

3.1.1 Distance and elevation

The relationship between the distance and elevation angle for the ground station and the satellite is shown in Figure 3.3 on the next page, and can be calculated the following way: The elevation angle El is found:

$$\cos(El) = \frac{rs \cdot \sin(\gamma)}{d} = \frac{\sin(\gamma)}{\left[1 + \left(\frac{re}{rs}\right)^2 - 2\left(\frac{re}{rs}\right) \cdot \cos(\gamma)\right]^{1/2}} \tag{3.1}$$

From Equation 3.1, the distance can then be calculated:

$$d = \sqrt{1 + \left(\frac{re}{rs}\right)^2 - 2\left(\frac{re}{rs}\right) \cdot \cos(\gamma)} \cdot rs \tag{3.2}$$

In Figure 3.3 on the next page the elevation angle is plotted versus the distance for a 400 km orbit height.

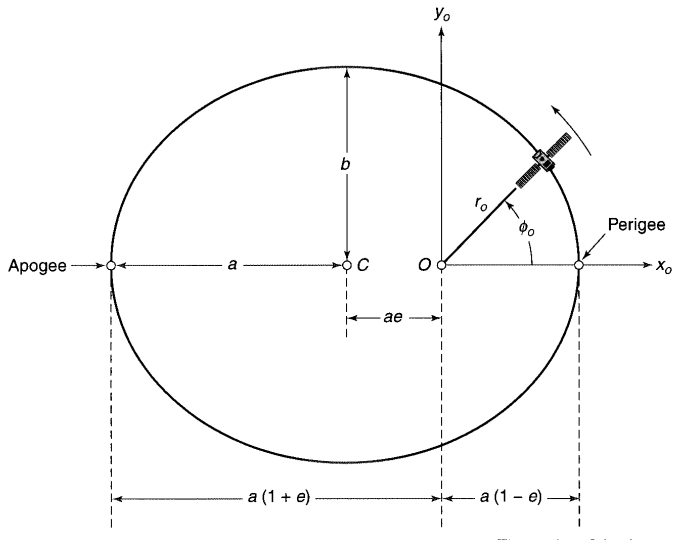


Figure 3.2: The orbit as it appears in the orbital plane, where O is the centre of the earth and C is the center of the ellipse. The relationship between the semimajor a and semiminor b axes defines the eccentricity e of the orbit.[13]

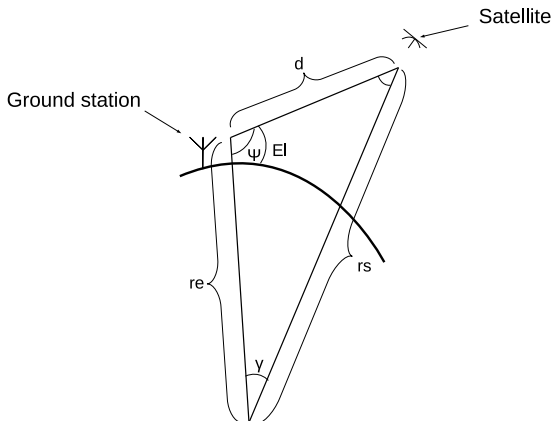


Figure 3.3: Relationship between distance and elevation angle between a ground station and a satellite.

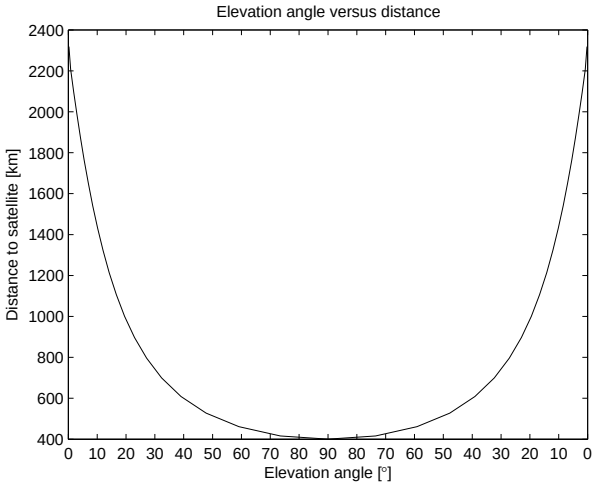


Figure 3.4: Elevation angle versus distance between a ground station and a satellite at a circular LEO of 400 km.

When modeling the satellite link, the distance and free space losses will dominate the degeneration of the signal strength. The link budget must therefore account for the worst case distance the satellite will have from the ground station.

[6] [13]

3.2 Link budget

The link budget is the output of a link analysis of a communication system. Through the calculation of gains and losses it is possible to obtain information about transmission and reception quality through noise sources, signal attenuation and other effects throughout the link, thus providing us a overall system insight. The information will expose if the system meets required specifications. [3][13]

I have made the following link budget calculation to illustrate what effects the considered data rates will have on the overall communication system, and how the results can be used to ensure that a reliable communication link can be achieved.

3.2.1 Calculations

The following calculations exposes what parameters are used in the link budget calculations, and how they affect the results.

$$P_r = \frac{P_t G_t A_e}{4\pi R^2} \quad \text{watts} \tag{3.3}$$

We are interested in finding the power at the receiving antenna. It can be expressed as in Equation 3.3, where:

P_t = The power at the transmitting antenna.
G_t = The gain at the transmitting antenna
A_e = The effective areal of the receiving antenna
R = The distance between the transmitting and the receiving antenna

Antenna gain can be expressed in therms of areal and wavelength.

$$G = \frac{4\pi A_e}{\lambda^2} \tag{3.4}$$

Inserting Equation 3.4 in the areal of Equation 3.3 gives us:

$$P_r = \frac{P_t G_t G_r}{(4\pi R/\lambda)^2} \quad \text{watts} \tag{3.5}$$

EIRP

Equivalent Isotropically Radiated Power (EIRP) is the figure of merit representing the transmitting antenna, and is the multiplication of power transmitted and antenna gain. The above formula can be represented as the general formula for power received:

$$\text{Power received} = \frac{\text{EIRP} \cdot G_r}{\text{Path loss}} \quad \text{watts} \tag{3.6}$$

Signal losses

The signal losses in the link budget can be categorized in losses within the transmitting and receiving antenna, signal attenuation due to atmospheric conditions, and the path losses related to the distance between the transmitter and receiver. The last one being the most decisive, and is described as

the free-space loss of electromagnetic waves. The physical effects of this is dependent on wavelength, thus higher frequencies have higher path losses.

$$P_r = \text{EIRP} + G_r - L_p - L_a - L_{ta} - L_{ra} \quad \text{dBW} \tag{3.7}$$

The power at the receiving antenna can together with the signal losses be expressed in logarithmic values as in Equation 3.7, where:

L_p = Path losses
L_a = Attenuation
L_{ta} = Losses within the transmitting antenna
L_{ra} = Losses within the receiving antenna

Noise temperature

$$\frac{C}{N} = \left[\frac{P_t G_t G_r}{k T_s B_n}\right]\left[\frac{\lambda}{4\pi R}\right]^2 = \left[\frac{P_t G_r}{k B_n}\right]\left[\frac{\lambda}{4\pi R}\right]^2\left[\frac{G_r}{T_s}\right] \tag{3.8}$$

The C/N calculation of the link budget, rewritten in terms of sender and receiver is shown in Equation 3.8. The G_r/T_s specifies the sensitivity of the receiving station and is a common figure of merit, T_s is the system noise temperature of the receiver. Noise temperatures are always expressed in kelvin.

The system noise temperature of the receiver is the total noise temperature generated in the receiver system. Calculating the noise temperature of a generic receiver with an antenna, a feed or waveguide and a Low Noise Block (LNB) can de done with Equation 3.9.

A ground station can only vary its antenna gain and system noise temperature to improve the SNR ratio. [13]

$$T_s = T_a/G_{feed} + (1 - G_{feed})T_{feed} + (NF - 1)T_0 \tag{3.9}$$

Where:

T_s = The total noise temperature
T_a = The noise temperature measured at the antenna
G_{feed} = The loss in the feed or waveguide as linear ratio
T_{feed} = The noise temperature of the feed or waveguide
NF = The noise figure of the LNB as linear ratio
T_0 = The reference noise temperature (290 K)

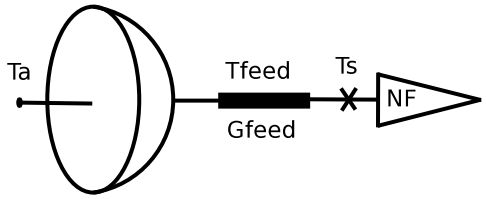


Figure 3.5: Noise temperature of a generic receiver.

We need these definitions to calculate the system noise temperature:

Noise temperature corresponding to a noise figure NF:

$$T_{nf} = (NF - 1) \cdot 290 \quad \text{K}$$

Noise temperature of a passive component of gain G:

$$T_p = \frac{1 - G}{G} \cdot 290 \quad \text{K}$$

Output noise temperature of a noiseless component of gain G:

$$T_o = G \cdot T_i \quad \text{K}$$

3.2.2 Eb/N0 and S/N relationship

$$C/N = E_b/N_0 \cdot \frac{fb}{B}$$

E_b/N_0 is the energy per bit to noise power spectral density ratio, and is a normalized version of the SNR or S/N. Both units are dimensionless, and are used to compare the quality between different communication systems. In digital communication it is without exception E_b/N_0 being used as a natural figure of merit. This way we can compare systems at bit level not taking account for the modulation technique. In digital signals it is more useful to characterize a signal by its energy, since a symbol is transferred by alternating a waveform in a time window, e.g. the symbol time. Digital signals are therefore energy signals, and have zero average power and finite energy. Analog signals with an infinite duration has finite average power and infinite energy, we then use the S/N measurement. [3]

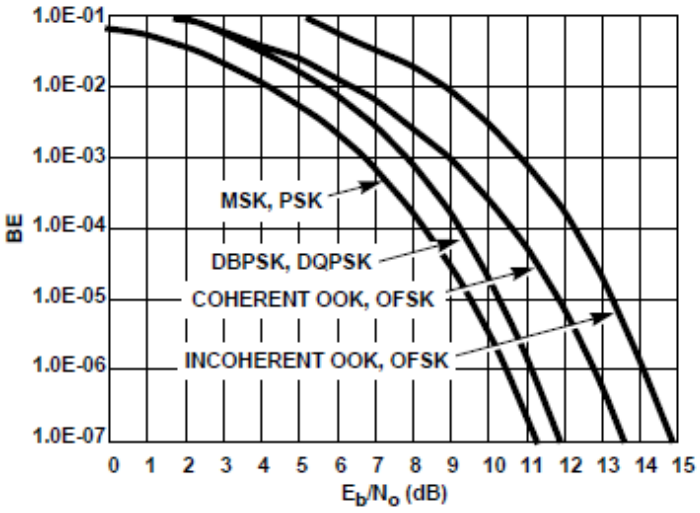


Figure 3.6: Probability of bit errors for common modulation methods.[14]

3.2.3 Eb/N0 and BER relationship

From the the results of a linkbudget we can obtain the $E_b/N0$ for the given communication system. By measuring that value against the waterfall plot in Figure 3.6, we can inspect the probability for bit errors to exist for different modulation methods. This way we can compare different modulation methods and inspect how liable our communication system is for errors.

3.3 Example for a LEO satellite

It is several ways of setting up a link budget. There exists commercially available software and spreadsheets that can be used. Noise temperature and propagation losses are often calculated differently from software to software, and spreadsheet to spreadsheet. ITU-R, the radio communications sector of ITU provides recommendations for propagation models. The Centre National d'Etudes Spatiales (CNES) have provided a library[1] of functions that can be used to calculate propagation losses. They have also provided a simple spreadsheet example for Microsoft Excel, that can take advantage of this library and functions.

I decided to use the spreadsheet example when setting up a link budget example for CubeSTAR and Oslo Ground Station. Due to version compat-

[1]More info about propagation models from CNES:
http://logiciels.cnes.fr/PROPA/en/logiciel.htm

ibility problems with Microsoft Excel and the library from CNES i decided to make the calculations in Mathworks Matlab[2], that is also supported by the propagation libraries. This simplified multiple calculations for various elevation angles and distances.

In Appendix A the input parameters together with the results calculated by MatLab are enclosed. An elevation angle of 10° that relates to a distance of 1432 km for an orbit height of 400 km have been used. The MatLab calculations are based on the equations deduced earlier in this chapter. A link margin of 15 dB is subtracted from the final results. We can from these results see that losses due to atmospheric losses can be neglected, because the orbital height of a LEO satellite and the frequencies used. The only losses that needs to be taken account for, are those introduced in the system itself together with the path losses.

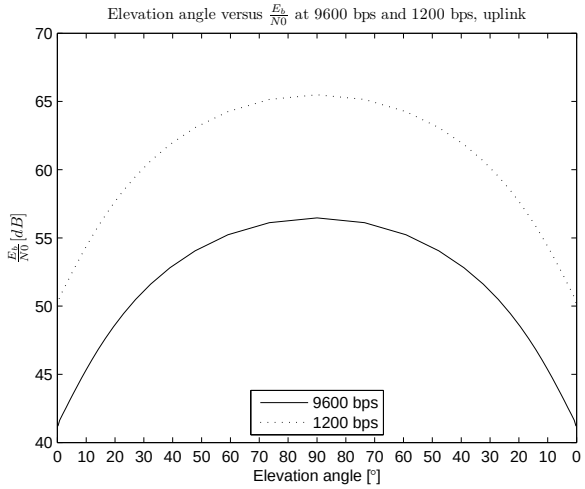
The calculated results for different elevation angles can be inspected in Figure 3.7 on the following page, and shows the differences in E_b/N_0 for both 9600 bps and 1200 bps. These are the bit rates supported by the AX.25 protocol and the ground station.

During a satellite pass over a ground station the elevation angle and the distance to the satellite varies. We must take this in account when designing the system. With the results we can find out how sensible the system is to BER. The worst-case-scenario will then need to be defined. A 0° elevation angle causes the longest distance, hence the highest path loss and lowest $E_b/N0$. The system could at the lowest elevation angles be influenced by noise from the earth, and the signal can be blocked due to the earths topology. Setting an elevation angle of 10° in the linkbudget calculations, makes an assumption that it is not possible to obtain a reliable communication for lower elevation angles.

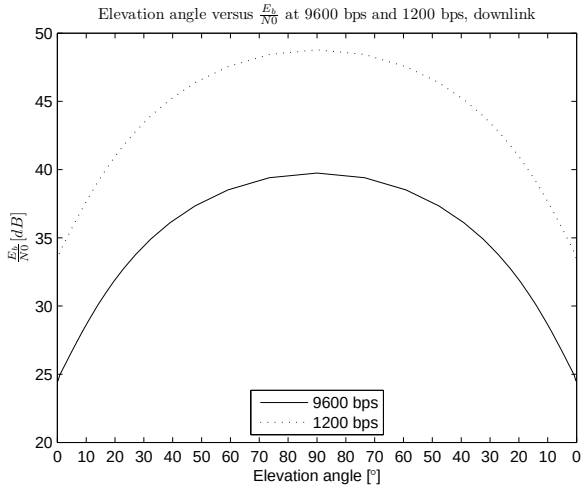
We can from Figure 3.7 on the next page read the E_b/N_0 for all elevation angles. The $E_b/N0$ can together with the modulation scheme be used to find the probability for bit errors in plots as seen on Figure 3.6 on the facing page.

The communication system can take advantage of the varying distance. In the downlink example, we can see that the $E_b/N0$ has a variation of 15 dB for 9600 bps between elevation angles $0 - 90°$. This can be utilized by reducing the power usage at high elevation angles to take account for the higher Eb/N_0. Adaptive coding and/or modulation can be used for achieving higher data rates at high elevation angles. These methods demands either an attitude system in the satellite capable of knowing exactly where in the orbit to trigger these changes, or a communication architecture where the

[2]More info about MathWorks MatLab:
http://www.mathworks.com/



(a) E_b/N_0 results for uplink budget.



(b) E_b/N_0 results for downlink budget.

Figure 3.7: Link budget results for various elevation angles at an orbital height of 400 km.
Comparison between bit rates of 1200 bps and 9600 bps.

ground station can reply with received SNR data.

Chapter 4

Communication architecture

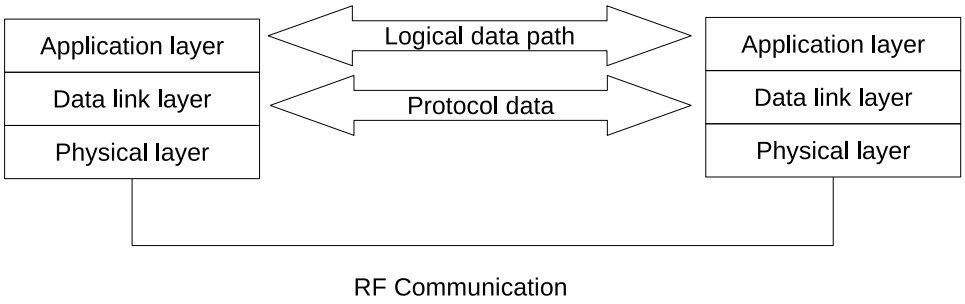


Figure 4.1: Simplified model for the layered communication architecture.

The Communication architecture is the arrangement, or configuration, of satellites and ground stations in a space system, and the network of communication links that transfer information between them. [12]

In the CubeSTAR mission, a store-and-forward network can be used. Telemetry data is stored in the satellite until an achievable communication link with a ground station is obtained, and then the data is forwarded. Support for Telemetry, Tracking & Command services must be ensured in the communication architecture, and a suitable protocol for handling this must be implemented.

Given this architecture, together with the services that CubeSTAR will offer, a simplified version of the OSI reference model is suitable for the CubeSTAR communication architecture. Figure 4.1 shows the abstraction layers in this simplified model.

A link layer protocol is the critical part in ensuring a reliable communication link for the telemetry data.

4.1 The data link layer

The data link layer should ensure the reliability of the physical link between two adjacent network nodes, like a point-to-point connection between a satellite and a ground station. This is performed by activate, maintain and deactivate the data link. The services provided to the higher layers is error detection and control. This is performed by the framing of the raw bit stream in the physical layer.

An ideal link layer protocol would provide a service of virtually error-free transmission over the link, to the higher layers. Packets with data from above layers are encapsulated in frames[1] before transmission. The frame consist of a header, the payload and a trailer.

The services provided by the physical layer is the modulating and demodulating process of the raw bit stream handled by the link layer, usually handled in the transceiver circuitry.[5][4]

Choosing a link layer protocol for the project must in addition to the above considerations, support already chosen hardware in the satellite and ground station node. In Chapter 5 the implementation of the protocol is described, together with the hardware setup.

I have chosen to implement AX.25 as the link layer protocol, and adapted it for this communication architecture. AX.25 is a supported communication protocol by the GENSO project.

4.2 AX.25 protocol and frame format

Bytes	1	14/21/28	8	8	0–256	16	1
	Head flag	Address	Control	PID	Data	FCS	Tail flag

Figure 4.2: Frame format for an AX.25 UI frame.

The AX.25 is a protocol based on High-Level Data Link Control (HDLC) frames, and an extension of the X.25 protocol for the support of amateur radio callsigns. In the written standard the protocol is introduced as an link layer protocol occupying the three lowest layers in the OSI reference model. This standard does not describe the physical layer, that has more or less became *de facto* from various implementations in the amateur radio community. The protocol provides services to the network layer for routing of AX.25 frames with higher layer protocols, e.g. TCP.

[1] In this paper the word “frames” are used consistent for the data in the link layer. In higher communication layers the notation “packet” is used.

To provide flexibility in the use of the protocol, AX.25 can be implemented and used in both a connection-oriented and a connection-less mode. The connection-oriented mode provides a reliable communication link in the way that the protocol itself ensures the proper opening and closing of a connection, regulating the communication flow and ensuring that any erroneous data should be attempted to correct.

AX.25 provides the connection-less mode with a special frame type called Unnumbered Information (UI) frames. This frame type allows any type of data payload and can be sent and received independently of other functionality in the connection-oriented operation mode.

The problem with using and implementing the whole AX.25 protocol stack with support for the connection-oriented mode, is that it is not well suited for being used to transport telemetry and commands to and from CubeSat spacecrafts. The protocol was originally developed to cover a switching network of several amateur radio nodes. Using the protocol for a two node store-and-forward network, as in a CubeSat mission, could shrink the informational data throughput more than necessary. The documentation of the protocol was not written for developers to implement the protocol, rather describing the usage.

It is possible to only implement support for the UI-frames. This simplifies the design of the protocol greatly, and provides full control of the implementation and how to adapt it to the mission. The drawback is that the protocol itself does not provide a reliable service to the higher layers, in terms of retransmissions and reassembling of frames to correct lost data, the custom implementation of the protocol or the application layer must take care of this. The data field of the UI-frame could be used to implement both custom protocol info and to carry the informational data.

[15]

4.2.1 Frame fields

There exists three general types of AX.25 frames:

1. Information frame (I frame)
2. Supervisory frame (S frame)
3. Unnumbered frame (U frame)

The rest of this paper will describe the special combinational UI frame, and how to implement the connection-less mode of AX.25. The AX.25 documentation describes the frame fields, but omits some of the theory that would be

Destination address							Source address						
A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14

Figure 4.3: Frame format for address fields without repeater addresses.

helpful in an implementation process. The frame fields for a typical UI frame is presented together with the size of the fields in Figure 4.2 on page 28. All fields but the Frame-Check Sequence (FCS) is transmitted Least Significant Bit (LSB) first.

Flag

The head and tail flag identifies the start and end of a frame. Both flags occupies one byte and have the same bit pattern, `01111110`. This characteristic byte can not occur anywhere else in the frame. If it should occur, bit stuffing is applied.

Address

The address field in all frame types can consist of minimum the address of the receiver and sender, and maximum two repeater address fields. AX.25 is a link layer protocol not indented for the routing of frames, however it is possible to relay frames over several stations from source to destination in a switching network of amateur radio stations.

The addresses refer to amateur radio callsigns and can be up to six ASCII characters long, upper case only. If the callsign is shorter than six character it is padded with ASCII white spaces. All characters in the address fields are left shifted one bit, this does not loose any information since the Most Significant Bit (MSB) in ASCII is not used.

Figure 4.3 shows the format of the address fields with a destination address and a source address. A1–A6 and A8–A13 are the address bytes. The address bytes are sent in the numbered order.

The last byte (A7 & A14) of each address field is the Secondary Station Identifier (SSID). This byte has the following bit pattern, where the right most bit is the LSB: CRRSSIDA, and the individual bits are as explained:

- *C*, identifies command and response frames, and differs between the senders *Source SSID C-bit* and the receivers *SSID C-bit*. If AX.25 prior to version 2.0 is being used, both bits are either set to "1" or "0". The purpose of the bits are to maintain proper control over the link during information transfer state. All frames are considered as a

command or response. In case of a command frame, the *Source SSID C-bit* is set to "0" and the *SSID C-bit* is set to "1", and vice versa in the case of a response frame.
If the preceding address is a repeater address, the C-bit will indicate whether the frame has been repeated and avoid repeating it again.
In the practical implementation only based on UI frames, this bit can be ignored and set to '0'.

- *RR*. are reserved bits, set to "11".

- *SSID*, an unique identifier for stations using the same callsign, can be set to a numerical value.

- *A*, the address extension bit. This bit is set to one when the preceding address is the last address in the address frame field. The receiver will then know if there are more addresse in the frame to read, or if to process the following bytes as the next frame field.

Control

The control field can be either one or two bytes long and maintains control information for the flow control of the transmission. The fields are being used in the information frame (I frame), supervisory frame (S frame) and the unnumbered frame (U frame).

For UI-frames the Control field is one byte long, and always has the value `0x03`.

PID

The Protocol Identifier (PID) field is in use only in the I and UI frames. This field identifies which layer 3 implementation is in use, if any.

By setting the field to `0xF0`, indicates that no layer 3 protocol is implemented.

Info

The info field can take an integral number of bytes between $0 - 256$ byte and are only allowed in certain frame types. The UI frame has support for the info field. This field can carry any kind of data, and is not evaluated by the protocol.

FCS

The FCS is a CRC consisting of two bytes, and is calculated by the sender prior to sending. It is important to note that the FCS is transferred in reverse bit-order, where the MSB is transferred first. All other frame fields are transferred LSB first.

The FCS is implemented as a CRC, consisting of 16 bits according to the CRC-16-CCITT standard.

The purpose of the FCS field is for the receiver to verify the integrity of the frame. When the receiver has successfully received one AX.25 frame, it calculates the same CRC check as the sender. If these FCS fields matches, the integrity is verified, hence no bit errors where introduced in the frame. The FCS field is an error detecting feature, and are not able to correct for transmission errors, only detect them. If the FCS fields does not match and bit error exists, the whole frame is dropped and the data is lost.[16]

4.2.2 Theory of CRC

The generator polynomial is the starting polynomial used to calculate the CRC. In the CRC-16-CCITT standard, it is of degree 16 and is $G(x) = 1 \oplus X(5) \oplus X(12) \oplus X(16)$

To calculate the CRC the sender appends 16 zero bits in the FCS frame field. The message $M(x)$ to be calculated is all the bits between the header flag field and the FCS field. All the following calculations are being done in modulo two, or in Galois Field $GF(2)$. The remainder from the following division is then being calculated: $r(x) = x^{16}M(x)/G(x)$. The frame to be transmitted before appending the trailer flag, will now be: $T(x) = x^{16}M(x) - r(x)$

The receiver then calculates $T(x)/G(x)$, the remainder of this division will be 0. If however $T(x)$ contains errors due to noise in the transmission, the remainder will differ from 0 and the receiver will detect error in the received frame. [5]

A common implementation of the CRC calculation in hardware, is done with the help of shift registers with linear feedbacks. Software implementations are based on this idea, but faster algorithms exists.

4.2.3 Bit stuffing

The frame header and trailer fields has the distinctive bit pattern of `01111110`, to avoid this pattern to exist anywhere else in the frame bit stuffing is used. Anytime the transmitter detects five contiguous ones in a row that is not

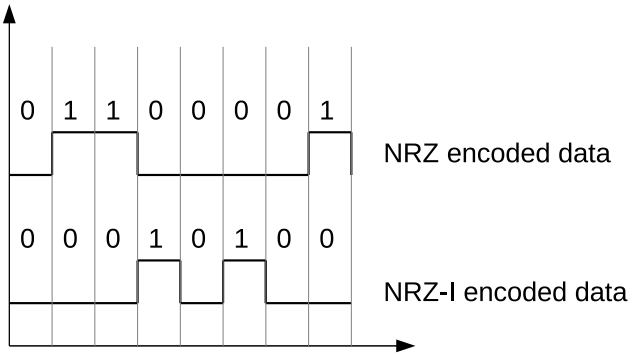


Figure 4.4: NRZ-I and NRZ encoding of a signal. In a NRZ-I encoded signal a 0-bit (space) is encoded as a change in transition level, and a 1-bit (mark) is encoded as no change in transition level.

a part of the head nor tail flag, it will insert a zero. Likewise when the receiver detects five contiguous ones in a row, it will remove any preceding zero. Hence the bit stuffing is applied in all AX.25 frame fields but the head and tail flag.

4.2.4 The physical layer

Although the AX.25 standard also defines the physical layer, this is not well documented. The physical layer is in transmitting responsible for the process of taking the raw bit stream from the frame and represent this as changes in voltage level for the transceiver to modulate. In receiving, it is the inverse process of translating voltage levels from the transceiver into raw bit data, that can be decoded according to the framing format. The standards for using the AX.25 over packet based amateur radio networks has become more or less *de facto*.

The 1200 bps and 9600 bps are two data rates that has become widely used standards and supported by the the amateur packet radio network.
Both transmissions rates take advantage of NRZ-I as line coding, inherited from the HDLC standard. In a NRZ-I encoded signal, each 0-bit (space) is coded as a change in transition level. Each 1-bit (mark) is coded as no change in transition level. Se Figure 4.4 for an example. This line encoding has the advantage that it is not the signal level that marks a bit, but the transition or the lack of transition that represents a bit. With the NRZ-I encoded signal the idea is that it will be easier for the receiver to synchronize and recover the clock frequency.

The modulation type used on 1200 bps transmissions is Audio Frequency Shift Keying (AFSK) according to the BELL-202 standard. This modulation

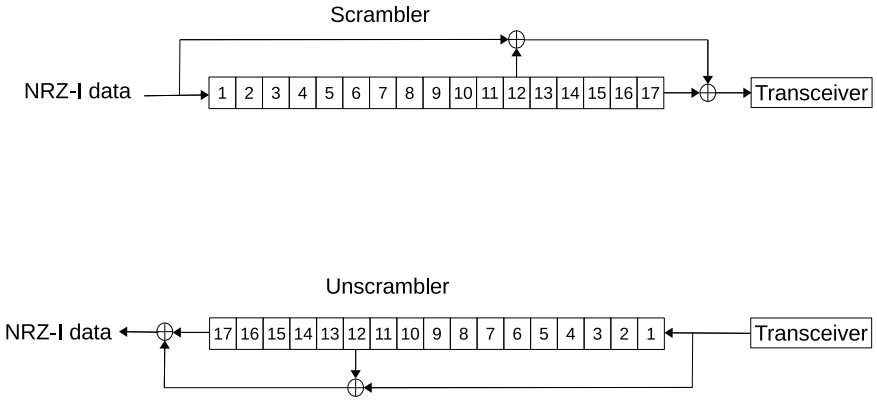


Figure 4.5: Scrambling functions in the 9600 bps G3RUH standard. The NRZ-I encoded data in the transmitter are scramled through a function so it appears pseudo-random. The receiving transmitter will use the same scrambling function to descramble the data and recover the NRZ-I encoded data.

type uses two audio tones for the mark and space bit, being 1200 Hz and 2200 Hz. Since AFSK is more or less an old way of modulating, common transceiver chips does not support this modulation type and the need for additional hardware to generate the audio tones become necessary. With standard amateur radio equipment it is however possible to generate AFSK from an Frequency Shift Keying (FSK) modulated signal. If transmitted at 1200 bps and FSK modulated with a frequency deviation of 1 KHz, is is possible to decode the signal as AFSK. The receiver needs to be capable of receiving in Single Side-Band (SSB) mode, it is then possible to get the 1200 Hz and 2200 Hz audio tones from the radio that can be directly fed into the TNC for decoding.

$$CBR = 2(\Delta f + f_m) = 2(3 \text{ KHz} + 4.8 \text{ KHz}) = 15.16 \text{ KHz} \tag{4.1}$$

The 9600 bps standard takes advantage of the FSK modulation type with a frequency deviation of $+/-3$KHz. It is common to use a gaussian filter in the baseband modulation process, for occupying less channel bandwidth and suppressing side lobes. The modulation is then called Gaussian Frequency

Shift Keying (GFSK). The 9600 bps standard was first described by James Miller. [17] The standard is widely called for the G3RUH standard [2] By applying Carson's Bandwidth Rule to the G3RUH standard, as in Equation 4.1 on the preceding page, we get a bandwidth of approximately 15 KHz.
A Scrambling function is used on the encoded data to avoid long runs of 0- and 1-bits, thus making the transmitted signal appear as random data. The scrambling process provides a more flattened RF spectrum in spreading the spectral energy. The scrambling polynomial is $X(Y) = X(1) \oplus X(12) \oplus X(17)$, where $X(Y)$ is the pseudo-random output. The current bit to be transmitted is a simple *exclusive-or* operation of the current bit and the bits sent 12-bits and 17-bits earlier. At the receiver side, the same polynomial is being used, where the current bit is *exclusive-or'ed* together with the bits received 12-bits and 17-bits earlier. See Figure 4.5 on the facing page for a representation of the scrambling functions.

Although pseudo-random data is being transmitted, the receiver will output the correct unscrambled data from the unscrambler function 17 bit delayed. It is therefore important that the shift registers in both sender and transmitter is synchronized with the same data. This can be achieved by sending 17 bits of fixed data before the AX.25 frame. As an example three additional head flags can be preceded the frame.
The reason that the shift registers has the non modulo eight length, is that it was originally designed to be implemented in hardware.

In packet based radio communication, transmission will be started with a preamble prior to sending the frames. This is a bit stream of variable length, with a pattern of alternating ones and zeros. This sequence ensures that the receiver clock will be synchronized, and for the receiver to lock on to the center frequency. Often the advised length of preamble is to be found in the datasheets of the transceiver circuitry. It would be best to leave the preamble unscrambled, so the alternating sequence is kept unaltered.

The process of generating and detecting an AX.25 frame for transmitting and receiving is represented as a flow chart in Figure 4.6 on the next page. Where in the flow the certain physical layer techniques are applied, can be inspected here.

4.3 The beacon signal

This chapter has now described how to implement a link layer protocol to support transfer of data for telemetry and commands. A different approach is used for tracking the satellite. Since the power budget in the satellite is

[2] The G3RUH name comes from the amateur callsign of James Miller, who have invented this standard.

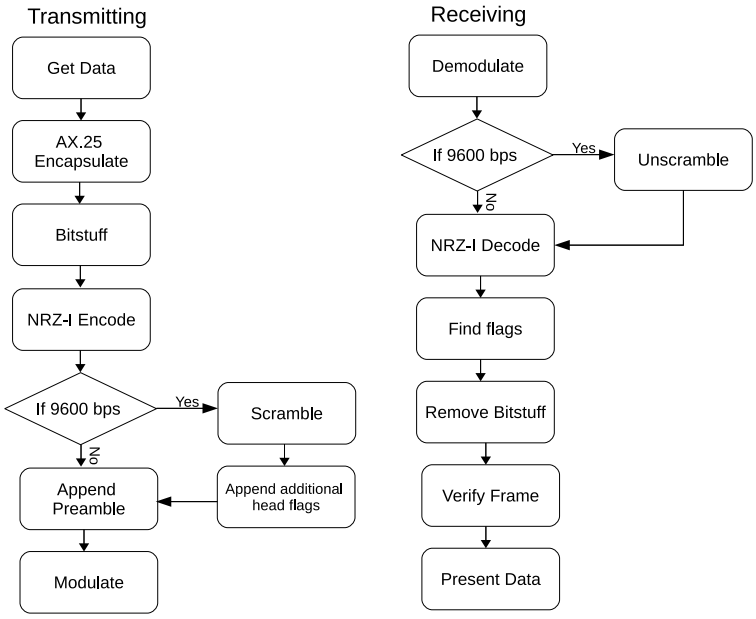


Figure 4.6: AX.25 frame processing flow chart.

limited, the communication sub-system will be in a low power mode until a command from the ground station wakes it up. During this low power mode, the satellite will transmit either continuously or at certain intervals a beacon signal. This signal is used to track the satellite.

In the beacon mode, hereby called Continuous Wave (CW) mode, the communication sub-system has the opportunity to work in a mode that does not pull as much power from the system as when in packet data mode. The data transmitted could be a morse encoded string containing the callsign of the satellite together with some additional housekeeping data. The morse encoded data can be modulated as a CW with an On-Off Keying (OOK) modulation. Software or the human ear can be used to detect and decode the morse coded beacon signal. Since the encoded data is modulated as a carrier wave being switched on and off, the receiving radio need to be tuned to a frequency where the demodulated baseband signal is represented as an audio tone. This can for the instance be achieved by using the SSB functionality of the radio.

A morse code beacon signal can be transmitted in various length and transmission rates. The transmission rate is expressed as “words per minute”, according to the “paris” standard that states how many times you can transmit consecutive morse coded signal “paris” in a minute. The transmission

rate should be kept at a level where it is possible to decode the signal by hart for an experienced radio amateur, although software most likely will be used.

4.4 The CubeSTAR protocol

As introduced earlier in this chapter, the AX.25 framing format is used in the data link layer implementation of the protocol. However the rest of the AX.25 protocol stack is omitted. Instead a custom protocol is implemented in the data field of the UI frame format of AX.25. This modification and implementation of the protocol is hereby called for the CubeSTAR protocol, This protocol must ensure that the following criterion are met:

- Satellite side node must accept commands from a ground station.
- Satellite side node must provide a service for the transfer of housekeeping data.
- Satellite side node must provide a service for the transfer of sensor data from the payload instrument.
- Satellite side node must transmit a beacon signal for the ground station to keep track of the satellite.
- Ground station node must provide a service for the transfer of commands to/from the satellite.
- Ground station node must provide a service for the receiving of sensor data from the satellite.
- Ground station node and satellite station node must detect and handle any lost or erroneous data.

The implementation of the CubeSTAR protocol consists of a framework where the above criterion can be utilized over AX.25 UI frames. This framework will then provide a service for the application layer in the satellite side and ground station node, so an integration with other sub-system in the satellite can achieve a reliable communication with the ground station.

At the satellite side node this application layer is the OBDH sub-system that is responsible for collecting and handling all data from various sensors and the satellite payload. The OBDH then communicates with the application layer at the ground station node through the link layer protocol.

The application layer at the ground station node can consist of a software application that through a graphical user interface communicates with the satellite and collects telemetry from the satellite payload.

The CubeSTAR protocol will need to be compatible with already chosen an existing hardware for the CubeSTAR project. The implementation and

realization of the CubeSTAR protocol is described in Chapter 5 on the facing page.

Chapter 5

The implementation

This chapter describes the hardware and software used in the development process of the CubeSTAR communication protocol. The implementation consists of a source code library and application written in C for the communication sub-system of the CubeSTAR satellite. I have also developed test applications for evaluating the communication protocol.

The development process consisted of a network of two nodes, the satellite node and the ground station node. Different hardware is used in both setups. The ground station node is an actual satellite ground station installed at the Department of Physics, UiO. The satellite node is a lab setup of evaluational and development kits of actual hardware chosen to be used in the communication sub-system.

5.1 Hardware setup for the satellite side

The hardware setup at the satellite side node consist of evaluation and development kits for the microcontroller and transceiver to be used in the communication sub-system. For the radio interface, a CC1101 transceiver from Chipcon is being used together with the CC1101DK433 development kit. The supplied software SmartRF Studio makes it an easy task to perform register settings in the transceiver and access all the features. The microcontroller used is an Atmel ATxmega 128A1. The development system STK600 and JtagICE MK2 from Atmel has been used to program the microcontroller and to interface with the transceiver evaluation kit. The setup and wiring of these kits can be seen in Figure 5.1 on the next page.

The transceiver chip has features for packet handling, and support for internal buffers. All data and transceiver configuration settings can then be transfered over a SPI bus between the transceiver and the microcontroller.

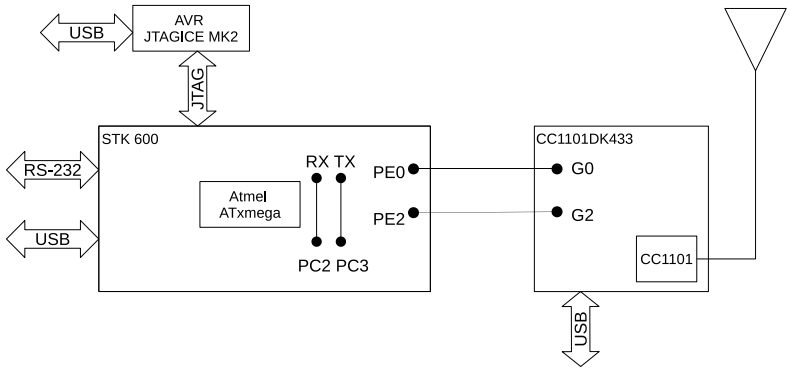


Figure 5.1: Development kits for the microcontroller and the transceiver.

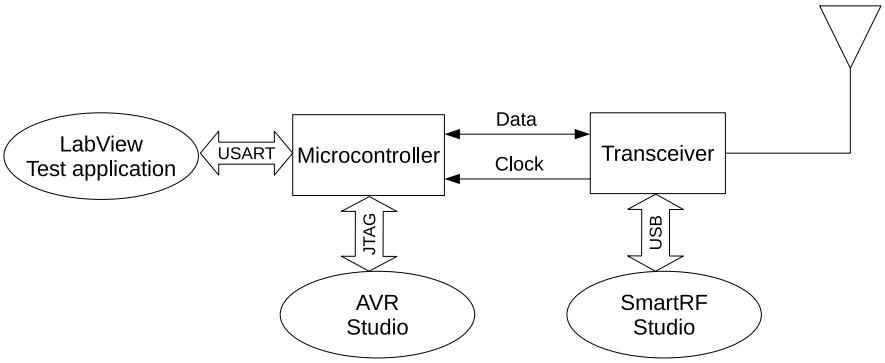


Figure 5.2: The hardware connected together for the development process of the communication protocol.

The transceiver can run in a "serial synchronous mode". Then the transceiver is transparent and the raw data bit stream is clocked in or out from the transceiver, depending on if it is configured in receive or transmit mode.

In the process of developing the communication protocol, the "serial synchronous mode" has been used. We can then eliminate possible problems related to the packet handling in the transceiver, since the microcontroller has full control of this. The communication software has been developed with the intention that it will be easy to exchange the "serial synchronous mode" with support for the "packet handling mode" in the transceiver. The motivation for using the packet handling features in the transceiver is to ease the microcontroller for some of the processing. When in receive mode, the microcontroller can be put in sleep, while the transceiver listens for a bit pattern to trigger on and detect a valid AX.25 header before waking the microcontroller.

In the "serial synchronous mode" the microcontroller and transceiver communicates over a simple two-wire serial bus, consisting of a clock and a data line. Since all transceiver configuration is done by SmartRF studio in this setup, there is no need for any transceiver configuration with the microcontroller at this stage. The serial bus is solely used for the binary data stream to be modulated when transmitting, or demodulated when receiving.

Design and implementation of the PCB and electronics for the communication sub-system, together with a Hardware Abstraction Layer (HAL) for all transceiver configurations is a part of J. Tresvig's masters project. [18]. The source code for the communication protocol will need to be integrated with that work at a later stage.

The development setup for the satellite side node can be seen in Figure 5.2 on the facing page.

5.1.1 Microcontroller

The Atmel ATxmega family was a new family of microcontrollers from Atmel when this project started. The small size and low-power technology makes experiences with the ATxmega microcontrollers interesting for this project.

Atmel has decided to change the programming style completely for the ATxmega family. It takes more advantage of structures in C and have better support for code portability between different ATxmega chips. All modules are defined in structures, i.e. all registers in ports, timers, ADC, DAC and so on, can be accessed by a single structure (`typedef struct`) for the specified module. The structure will be the same for all the ATxmega A devices. [19]

Where Atmel described the usage of the modules with source code examples

in the manuals and datasheets of the earlier microcontroller families, they have changed this for the ATxmega. The manual for a specific ATxmega chip does not fully describe the usage for the different modules, instead this is described in different application notes, e.g. application note for SPI, USART, Clock. The application notes include driver files with examples of the usage. These driver files can consist of several files of source code containing a "thin wrapper" of the actual code that performs the wanted actions.

As an example, enabling the external oscillator as the clock source was done by simple fuse settings in earlier chips prior to the ATxmega. Now this is done in the actual source code. The clock driver supplied with the application note can be used, but it will be more convenient to break it down and only use the actual source code as presented in Listings 5.1.

Listing 5.1: Enabling external oscillator as clock source in the ATxmega

```
1 /* Enable external oscillator as clock source */
2   void init_clk()
3   {
4       OSC.XOSCCTRL = OSC_FRQRANGE_2TO9_gc |
            OSC_XOSCSEL_EXTCLK_gc;
5       OSC.CTRL |= OSC_XOSCEN_bm;
6       while ( (OSC.STATUS & OSC_XOSCRDY_bm) == 0);
7       CCP=0xD8;
8       CLK.CTRL = CLK_SCLKSEL_XOSC_gc;
9       OSC.CTRL &= ~OSC_RC2MEN_bm;
10  }
```

5.1.2 Transceiver

The CC1101 transceiver can operate according to the standards for sending AX.25 data at 9600 bps. The FSK modulation with gaussian filtering is supported with frequency deviation of $+/-3$ KHz. Bit rate of 9600 bps is supported. The lowest frequency deviation supported is 1.6 KHz. With the lack of support for AFSK modulation it is not possible to use the 1200 bps standard for AX.25, even in SSB mode as described in Section 4.2.4 on page 33. The support for OOK modulation in the transceiver can be used for transmitting a morse coded beacon signal. [20][21]

The transceiver is by SmartRF Studio configured with the following settings (see Figure 5.3 on the next page):

1. The RF output level can be adjusted in the following steps: -30, -20, -15, -10, -5, 0, 5, 7, 10.

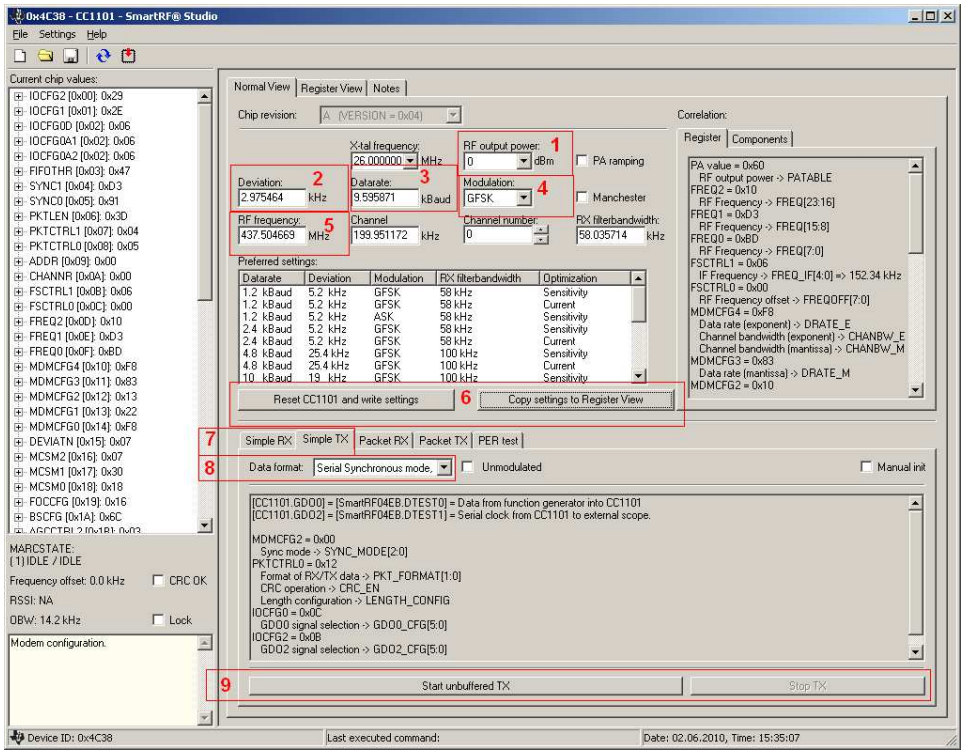


Figure 5.3: SmartRF Studio screenshot. The software is used to configure all transceiver settings.

2. The deviation is the frequency shift from the center frequency and is the same for both sides. This value must be 3 kHz for the +/- 3 kHz deviation in the 9600 bps (G3RUH) standard.
3. The datarate is set to 9.600 kBaud. Symbol rate being one bit per symbol.
4. The modulation is GFSK.
5. The frequency to be used. In this example, one of the frequencies of the NCUBE-2 satellite.
6. The setting are written to the transceiver here, and the actual values can be inspected in the Register View.
7. Simple RX and Simple TX was used for testing. These functions does not use any of the inbuilt packet features in the transceiver.
8. The data format is "Serial Synchronous mode". This way the transceiver uses two wires for the serial interface with the microcontroller. One being the data line and the other one is the clock from the transceiver. The clock rate will be the same as the datarate set in 3.
9. The start and stop buttons is the same for Simple RX and Simple TX, and as described starts and stops the transmission or reception.

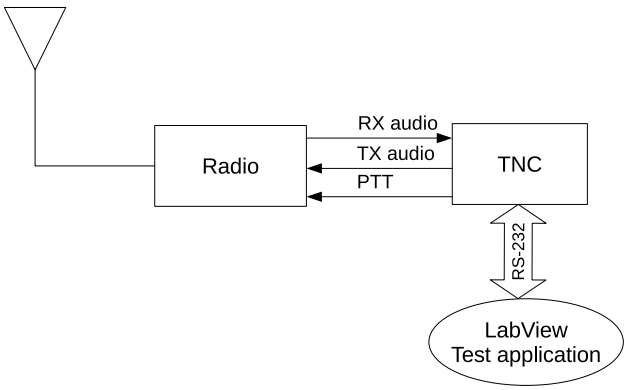


Figure 5.4: The hardware for the ground station communication system.

5.2 Hardware setup for the ground station

The CubeSTAR project have set up a GENSO compatible ground station, the Oslo Ground Station. It consist of a steerable antenna rig and an Icom IC-910 radio. With a computer connected to this rig it is possible to track and communicate with CubeSats. The radio has a data interface that can communicate with a TNC. The antenna rig and radio connection has been a part of a masters project by Henning Vangli[22].

In a packed based communication network, a TNC is responsible for the baseband modulation/demodulation and AX.25 encoding/decoding.

The data interface at the radio provides two audio lines for transmitting and receiving together with a PTT line to key the radio.

5.2.1 Terminal Node Controller (TNC)

The TNC works like a modem and is responsible for encoding any data to an AX.25 frame, and provide a raw bit stream for the radio to modulate and transmit. When receiving, the TNC is responsible for decoding the received bit stream into an AX.25 frame. Figure 5.5 on the facing page illustrates this process. The decoded or unencoded data is called a HDLC frame. Common

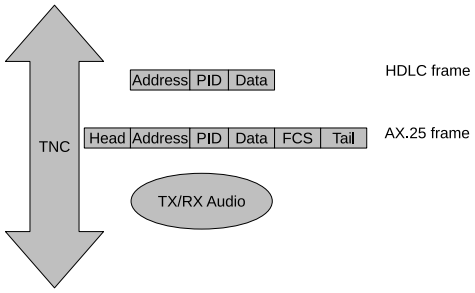


Figure 5.5: The basic operation of a TNC.
Converting between baseband audio and HDLC frames.

for all TNC's is that they can communicate with the outside world through a serial interface (RS-232 or USB).

I decided to use the MixW software TNC for this project, and chose it for its simple user and configuration interface. Since MixW is a software TNC, it was easy to evaluate and test it without the need for additional hardware. A hardware TNC was not available at the Oslo ground station and has not been tested, but could easily replace MixW if needed.

Because the lack of a hardware TNC and components for sending AX.25 data with the Oslo Ground Station, and because the lack of licensed radio amateurs capable of operating the rig, the transmitting of AX.25 commands with the Oslo Ground Station has not been performed. The transmit and PTT lines in Figure 5.5 are thereby not in place.

KISS TNC

To allow full control of the TNC for the user and/or users application, it is possible to run the TNC in KISS [1] mode.

A KISS TNC is a simple asynchronous protocol a TNC can operate in. The TNC does no protocol handling in this mode and works as a transparent device.

[1] KISS is a well used design principle and is an acronym for "Keep it simple and stupid" or "Keep it short and simple". The philosophy is that unnecessary complexity should be avoided, and the key goal should be simplicity.

Abbreviation	Description	Hex value
FEND	Frame End	0xC0
FESC	Frame Escape	0xDB
TFEND	Frame Escape	0xDC
TFESC	Transposed Frame Escape	0xDD

Table 5.1: Special characters in the KISS TNC protocol.[23]

When receiving, the TNC takes care of the decoding of AX.25 frames. The CRC sum is calculated to verify frame integrity, before the frame is transfered over the serial interface to the host. Both tail and head flags are together with the *FCS* field stripped away. This frame is called for a HDLC frame. The frame is preceded with start bytes and appended with stop bytes to mark the start and end of a frame.

Transmitting a frame through the TNC in KISS mode is the reverse process. The HDLC frame sent to the TNC containing *Address*, *PID* and the *Info* data. The frame is preceded and appended with start and stop bytes, before converted to an AX.25 frame in the TNC.

Table 5.1 shows the special characters used in the KISS TNC protocol. Because the communication architecture is only based on AX.25 UI, frames it simplifies the usage of the KISS TNC protocol. The start of a frame is preceded with the FEND byte `0xC0` and the zero byte `0x00`. The end of a frame is preceded with the FEND byte `0xC0`. If the FEND byte should occur in the HDLC frame, the FEND byte is translated in to the two byte sequence FESC TFEND. Similar if the FESC byte should appear in the user data of the HDLC frame, it is replaced by the two byte sequence FESC TFESC. [23]

Commercially available TNC's usually have the KISS TNC protocol implemented, but needs to be configured to use it.

MixW is set up to emulate the KISS TNC protocol, and can then then communicate through a serial port on the computer. To send and receive HDLC frames with the test application developed for the ground station it must be gained access to this serial port by software. A serial port bridge between the emulated serial ports COM5 and COM6 was set up, and MixW was set up to emulate KISS TNC over serial port COM5. Virtual Serial Ports Emulator by Eterlogic[2] was used to assign a virtual serial port, so other software could gain access to MixW over the KISS TNC protocol.

[2]More info about the Virtual Serial Ports Emulator from Eterlogic.
www.eterlogic.com

5.3 Software development

The software I developed for the communication sub-system consists of an independent library for the encoding and decoding of AX.25 frames, and the main program that uses the AX.25 library for packet handling and other control of the communication sub-system. A HAL for controlling the microcontroller, transceiver and busses can be included in the main program. The AX.25 library is written to be independent of microcontroller and transceiver, so that these can be replaced without big intervention. This will be of advantage for other projects that could have use of implementing support for AX.25 UI frames, e.g. other CubeSat projects. Together with the main program, the software and the HAL, the communication sub-system can be optimized for the ATxmega microcontroller and the packet handling features of the CC1101 transceiver, when they are to be integrated on the communication sub-system PCB.

I have created the following files in this project:

- `main.c` – Main communication sub-system program.
  Source code in Appendix B, Listing B.1 on page 91.
- `main.h` – Header file for the communication sub-sysem program.
  Source code in Appendix B, Listing B.2 on page 106.
- `debug.c` – Functions for transmitting debug signals.
  Source code in Appendix B, Listing B.3 on page 108.
- `debug.h` – Header file for `debug.c`
  Source code in Appendix B, Listing B.4 on page 110.
- `ax25.c` – Library for AX.25 packet handling.
  Source code in Appendix B, Listing B.5 on page 110.
- `ax25.h` – Header file for AX.25 library.
  Source code in Appendix B, Listing B.6 on page 126.
- `test_ground.vi` – LabView test program for the ground station.
- `test_xmega.vi` – LabView test program for the microcontroller.

5.3.1 Program flow

The main program is structured into three different states, *CW State*, *RX State* and *TX State*. These are the states the communication sub-system can operate in, and from each state it is possible to call the HAL for configuring the transceiver and microcontroller. Figure 5.6 on the following page illustrates the flow of the program.

After default initializations of variables and functions, it is possible to configure standard settings for frame transmitting. The amount of preamble bytes, additional flag bytes and delay between frames can be set. The *Sender* and

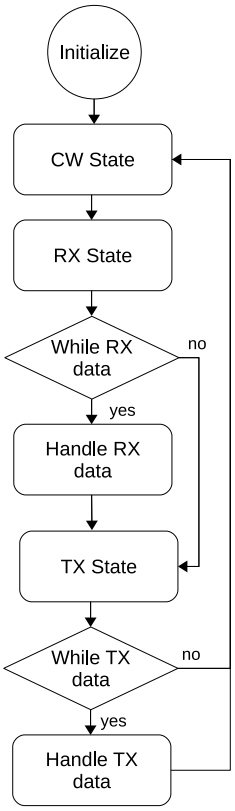


Figure 5.6: Program flow of the communication sub-system.

Receiver addresses together with the *PID* field in the AX.25 frame can be set. These fields are then NRZ-I encoded and scrambled before put in transmitting buffer. The *FCS* is pre-calculated for these fields. These data will be fixed for all frames to transmit, so by storing this it will ease processing time.

Once initialized, the program will start up in the *CW State* where the beacon signal can be implemented. A implementation of the beacon signal is not present, as this will only consist of functions in the OBDH calling functions in the HAL. The *CW State* can be woken by generating a timer interrupt or equivalent. The purpose of the *CW mode* is for the ground station to tune in to the correct frequency and to verify tracking of the correct satellite. The time that the *CW mode* transmits must be long enough that this can be achieved without problems. After the *CW State* has performed its action for a given time, the *RX State* is woken. The *RX State* will run continuously, until interpreted with received data that will need to be decoded, or data is to be transmitted.

The *RX State* will poll for a valid flag in a RX data structure in the AX.25 library. If valid data is detected, functions in the AX.25 library have decoded the data, and the frame fields can be read. In this implementation the received data is sent over the USART in a KISS format, when valid data is detected. The decoded data fields are directly available.

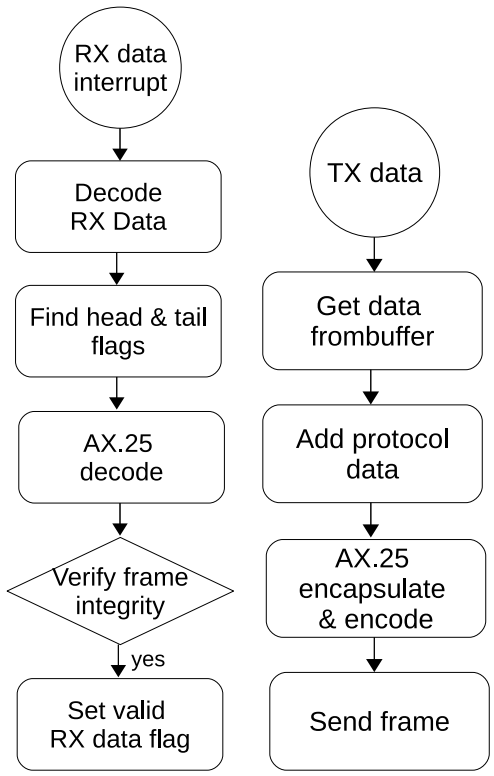
The program continues to the *TX State*, and the TX data buffer is checked for new data. All data present in the buffer is then transmitted immediately. After transmitting, the program will fall back to *CW State*. In Figure 5.7 the process of reception and decoding, and transmitting and encoding is illustrated.

In the final integration, the time for how long the *CW State* and *RX State* should run must be evaluated, together with when *TX State* should transfer the buffered data. The communication sub-system will have a half-duplex connectivity because uplink and downlink uses the same frequency bands, and transmission in both directions simultaneously could cause interference. The program flow is thereby structured this way.

5.3.2 The AX.25 library

The AX.25 library consists of functions for the encoding and decoding of AX.25 frames. Two different structures are used to retain all encoded and decoded data. The `(typedef struct TX_FRAME_struct)` and the `(typedef struct RX_FRAME_struct)` are used to store data for the AX.25 frame fields before encoding to transmit, and after a decoded and received frame.

The `(typedef struct TX_BUFFER_struct)` is used to store the en-



(a) Data reception and decoding. (b) Data transmitting and encoding.

Figure 5.7: Flow chart of data handling.

coded data to be transmitted. The data is AX.25 and NRZ-I encoded, and G3RUH scrambled. The scrambling can be deactivated for supporting other standards. It is possible to buffer preamble and additional data prior to the frame in this buffer. The size of the transmit buffer must be set to be big enough in the header file.

In the `(typedef struct RX_BUFFER_struct)`, the NRZ-I decoded and unscrambled data is stored. This data can then be AX.25 decoded, and frame integrity can be checked, before storing the decoded frame fields in the `(typedef struct RX_FRAME_struct)`. All frame fields can then be accessed by this structure, and optionally functions for filtering received frames on *PID* and *Address* fields can be added.

5.3.3 Buffers

The main program contains buffers for received data and data to be transmitted. Both buffers are FIFO's, implemented as ring buffers. Both buffers take elements of 256 byte. An illustration of an half-full ring buffer of 8 elements is shown in Figure 5.8 on the next page.

In the `(typedef struct TX_RING_BUFFER)` packets can be stored packets prior to AX.25 encoding. The OBDH can call the `create_frame()` function for buffering packets in the transmit buffer. The element size is limited to 256 byte since this is the maximum size of the info field in the AX.25 UI frame. The packets stored in this buffer contains the packet identificator and the informational data.

The `(typedef struct RX_RING_BUFFER)` can be used for the integration with the HAL for buffering received frames when operating the transceiver in "packet handling mode". The size of each element is limited to 256 bytes, but can be changed to whatever is convenient for the transceiver. This buffer is not used in the "serial synchronous mode".

5.3.4 Packets

The main program for the communication sub-system can take advantage of the info field in the AX.25 UI frames by using the library I have developed for implementing the informational data and additional protocol data for the CubeSTAR protocol.

I have used one byte as a packet identificator for the CubeSTAR protocol. The `(enum commands)` contains these identificators, and can be used to add other packet types. I have implemented the following packets as an example:

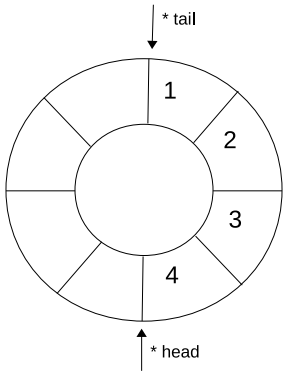


Figure 5.8: An example of a half-full ring buffer, of size 8. This buffer type is used for storing packets prior to AX.25 encapsulating and encoding, and received unencoded data. By adjusting the head and tail pointers it is possible to retransmit data. Hence ARQ techniques can be implemented here.

- COMMAND – Can be used for commands from the satellite.
- HOUSEKEEPING – Can be used for housekeeping data.
- TEST – Used for testing the protocol efficiency, contains sequence number.
- PAYLOAD – Can be used for data from the payload sensor, contains sequence number.
- RAW – Inserts no packet identicifator, can be used to transmit raw data in the info field, e.g. ASCII text.

Each packet identificator is a byte of two nibbles, where the most significant nibble is used for the identificator, and the least significant is used for sequence numbering in the packets where this is implemented. The sequence number is of four bits and will overflow after `0x0F`. Thus making 16 individual sequence numbers. The `(enum commands)` can be expanded to include other packet identificators and the packet identificator byte it self can carry more information. Various ACK or NACK messages and other commands can be implemented here.

5.3.5 Error detection and handling

The communication protocol on the satellite side can detect errors by taking advantage of the *FCS* field in the AX.25 frames. Since the protocol is implemented as AX.25 UI frames, there exists no flow control by the protocol itself in how to handle lost data.

For the telemetry an optional sequence number can be added to the packets. The ground station will then have the opportunity to detect if frames where lost on its way. Because the transmit buffer at the satellite side buffers all sent packets, it is possible to retransmit some or all of them. Functions for performing this are implemented. By using the ring buffers in the main program for buffering sent packets, it is a straightforward task to change position of the `(uint8_t *tail)` pointer for setting what packets to retransmit. This can be used to implement common ARQ modes, or other custom retransmission modes that will be needed in a final design.

5.3.6 Debugging

For making it easier to debug errors and to get feedback from the microcontroller to the test applications, the debug files `debug.c` and `debug.h` are included for sending commands over the USART interface. The AX.25 library uses functions in these files for sending commands whenever frames are detected, and if there should be a mismatch in the CRC calculation of

(a) Example of an unencoded frame

Offset	0x00	0x10	0x20
0x00	0x7E	0xA4	0x33
0x01	0x7E	0x61	0xF9
0x02	0x7E	0x03	
0x03	0x7E	0xF0	
0x04	0x8A	0x48	
0x05	0x82	0x65	
0x06	0xA4	0x6C	
0x07	0xA8	0x6C	
0x08	0x90	0x6F	
0x09	0x40	0x20	
0x0A	0x60	0x77	
0x0B	0x86	0x6F	
0x0C	0x84	0x72	
0x0D	0xA6	0x6C	
0x0E	0xA8	0x64	
0x0F	0x82	0x61	

(b) Example of a NRZI-encoded frame

Offset	0x00	0x10	0x20
0x00	0x7F	0xC9	0x02
0x01	0x7F	0x75	0x01
0x02	0x7F	0x54	0x01
0x03	0x7F	0x05	
0x04	0xD3	0x6D	
0x05	0xD4	0x76	
0x06	0x36	0x71	
0x07	0xCD	0x71	
0x08	0xDA	0x70	
0x09	0x6A	0xB5	
0x0A	0x75	0x87	
0x0B	0xD7	0x8F	
0x0C	0xD6	0x84	
0x0D	0xC8	0x8E	
0x0E	0x32	0x76	
0x0F	0x2B	0x44	

(c) Example of a G3RUH scrambled and NRZ-I encoded frame

Offset	0x00	0x10	0x20
0x00	0x7F	0x80	0x1E
0x01	0x8F	0xD1	0xBE
0x02	0x76	0x4D	0x00
0x03	0x09	0x7B	
0x04	0xA9	0x42	
0x05	0x56	0xA7	
0x06	0x0E	0x81	
0x07	0x85	0x25	
0x08	0x96	0x2B	
0x09	0x08	0x4C	
0x0A	0xD1	0x13	
0x0B	0xD6	0x23	
0x0C	0x19	0x93	
0x0D	0xF8	0xFA	
0x0E	0x80	0xF9	
0x0F	0xD4	0x2E	

Table 5.2: Example of data in the encoding process of a frame.

the *FCS*. From the AX.25 library it is possible to output signals from received data. The unscrambled signal and the NRZ-I decoded signal can be output on specified pins on the microcontroller. This way it is possible to debug received data, if it is not decoded properly.

The main program passes messages of the program flow to the test application over an USART interface.

5.3.7 Example of a frame

The example provided in Table 5.2 is an AX.25 UI frame, where the sender address is "CBSTAR", the receiver address is "EARTH " and the *Control* and *PID* field is `0x03` and `0xF0`. The ASCII message in the info field is "Hello world". As seen on Table 5.1(b) and 5.1(c), the length of the data appears to be one byte longer than the original data. This is due to bit stuffing applied in the *FCS* field, and only the LSB of the last byte is a valid bit.

5.3.8 Memory usage

The memory consumption in the microcontroller is as listed from compiling in AVR Studio:

```
AVR Memory Usage
----------------
Device: atxmega128a1

Program: 11676 bytes (8.4% Full)
(.text + .data + .bootloader)

Data: 6072 bytes (74.1% Full)
(.data + .bss + .noinit)
```

Code optimizing and evaluating the buffer sizes in both the main program and the AX.25 library according to the actual packets that will be used can decrease the amount of memory used.

By introducing external memory it is possible to store data from the ring buffers in the main program used to buffer packets, outside the microcontroller. This decreases the memory usage and raises the opportunity to implement bigger buffers, and thereby higher capacity to store data for possible retransmissions.

5.3.9 Integration with transceiver packet handling mode

By using the "packet handling mode" of the transceiver, it is possible to keep the processing in the microcontroller at a lower level when in *RX mode*. The transceiver will be responsible for detecting a valid start of an AX.25 frame. The drawback is that additional delays are introduced in *TX mode*, when encoding whole packets in the microcontroller and buffering them for the transceiver to transmit.

The communication sub-system that the communication protocol will run on, is structured as in Figure 5.9 on the next page. In Section 5.3.3, the buffers in the main program that can be used in this final integration are described. The services offered to the HAL controlling the SPI communication, are a buffer of AX.25 encoded data ready for transmitting. The HAL can buffer received data for the communication protocol to process and decode. Since commands to CubeSTAR always will have a fixed header when the frames are scrambled due to the known scrambling polynom, it is possible for the

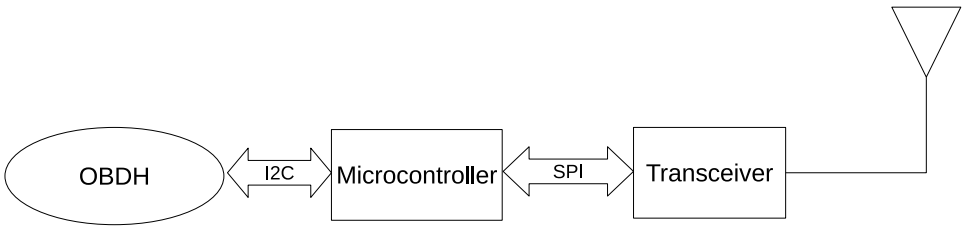


Figure 5.9: The future integration of the communication sub-system.

transceiver to use either the whole header or a portion of it to trigger on. This scrambled header can be found by inspecting the TNC that will be used at the ground station.

The services offered to the OBDH are the decoded frame fields of received data, and the FIFO for informational data to transmit. These buffers and structures are described in Section 5.3.3.

5.4 Graphical test interface

For testing the communication protocol, two applications in LabView[3] was developed. LabView was chosen for its advantage of being scalable, and powerful for representing data graphically. Although the tests for evaluating the protocol efficiency does not need any LabView specific components, it is convenient to have a LabView module for this testing. At a later stage these test programs could be adapted to include other tests of the communication system.

The test applications can send data to be encoded, and can represent received data encoded in KISS format. Initiating tests for measuring the efficiency and representing the results are possible from the test applications. These tests are presented in Chapter 7.

5.5 Transmitting and receiving

When testing that the software together with the transceiver decodes and encodes frames properly, a oscilloscope was used. The debug lines from the microcontroller together with clock and data lines connecting the microcontroller and transceiver was probed.

[3]More info about LabView from National Instruments:
http://www.ni.com/labview/

For verifying that transmitting works, the Oslo Ground Station at the UiO was used. This test setup is explained in Chapter 7. Experimenting with different lengths of preamble, a length of 56 bytes was found to be long enough for the ground station to receive and decode all sent frames. Preamble length measured by the oscilloscope is seen in Figure 5.12 on page 59. The length of a fully utilized frame, 256 byte in the info field, can be seen in Figure 5.13 on page 59. The processing time in the microcontroller for generating an encoding an AX.25 UI frame, is based on the length of the frame. A fully utilized frame will have a delay, measured between two consecutive frames, of 11.25 ms. Seen on Figure 5.11 on the following page.

Transmitting with the CC1101 configured by SmartRF Studio according to the G3RUH standard for 9600 bps, discovered that the transceiver transmitted on a frequency 100 KHz above the one specified.

Verifying the correct reception of commands as AX.25 data was done by a CanSat[4] kit from Pratt Hobbies. This kit is capable of sending AX.25 UI frames at 9600 bps, G3RUH scrambled. Configuration of the CanSat kit is done by a simple Arduino[5] interface. It is possible to change frequency, 1200/9600 mode, and set the *Addresses* together with the contents of the *Info* field. A delay before transmission can be set. More advanced communication is not available with the CanSat kit because of the simplex connectivity. Strange things occured when transmitting with delays between frames lower than 200 ms. The data in the *Info* field would not always be correct.

When receiving data it is possible to route the unscrambled and NRZ-I decoded data out on a debug port from the microcontroller, and thereby ensure that the frames are received correctly. This was of great value when investigating that the descrambling and NRZ-I decoding worked correctly. In Figure 5.14 on page 60 the received data sent from the CanSat kit is presented in the oscilloscope plot.

The transmitting and reception was verified at the Andøya Rocket Range (ARS) ground station, where a Kantronics KPC-9612+ TNC is used.

In Figure 5.10 on the following page the time for transmitting an fully utilized AX.25 UI frame at 9600 bps, is presented.

[4] More info about the CanSat kit:
http://www.pratthobbies.com/proddetail.asp?prod=CANSAT-1

[5] More info about the Arduino programming interface:
http://www.arduino.cc

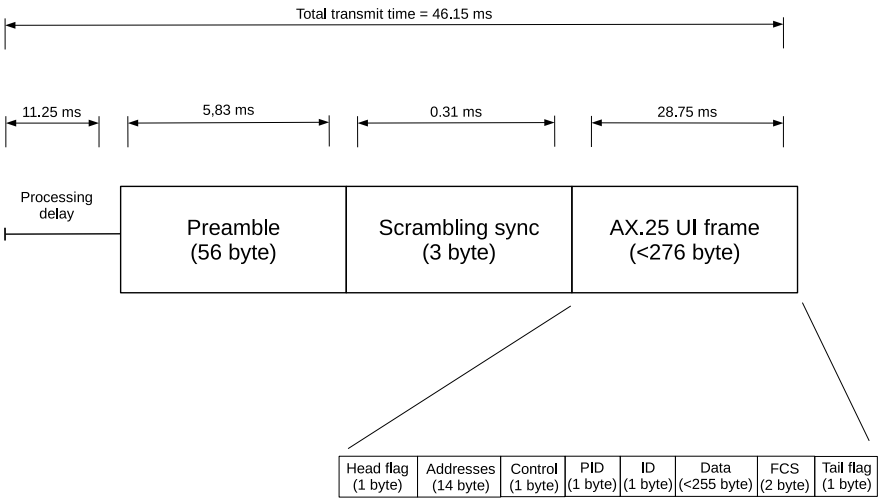


Figure 5.10: The time for one frame with a fully utilized info field to transmit at 9600 bps.

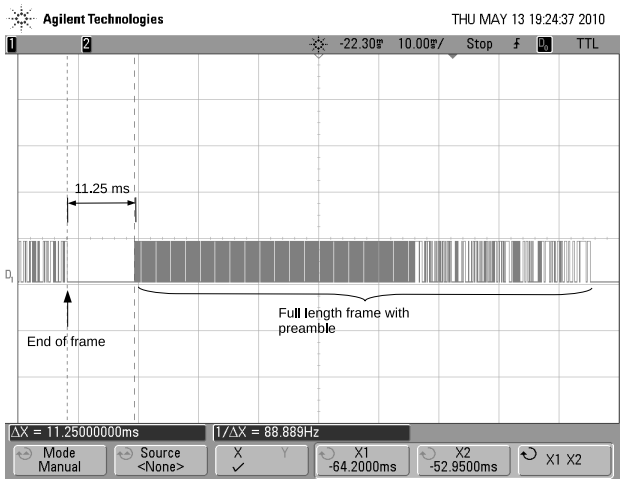


Figure 5.11: The processing delay when transmitting, measured between two consecutive frames.

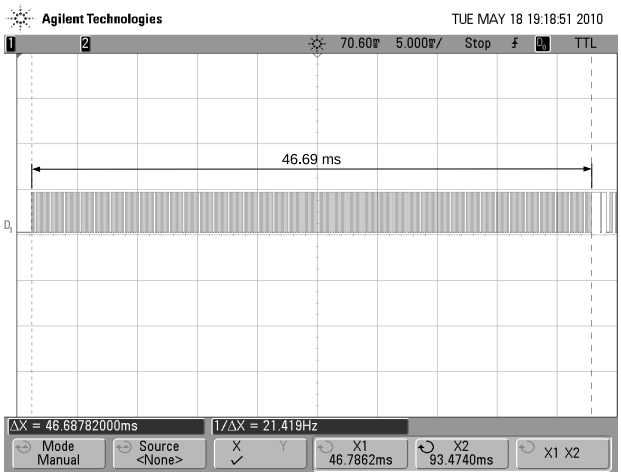


Figure 5.12: The preamble length, measured before a transmitted frame.

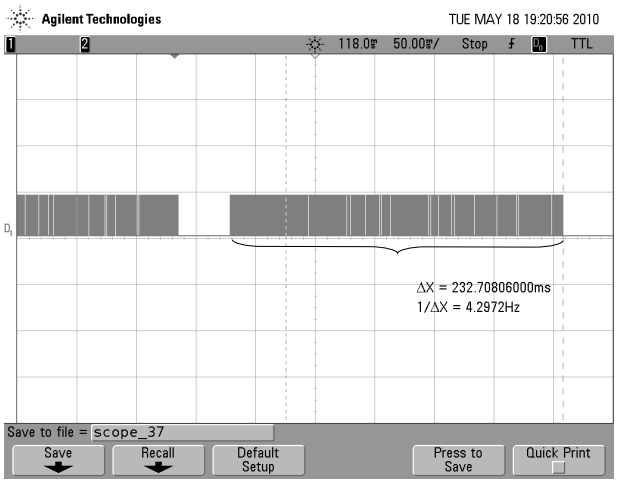


Figure 5.13: The measured length of a transmitted frame with a fully utilized info field.

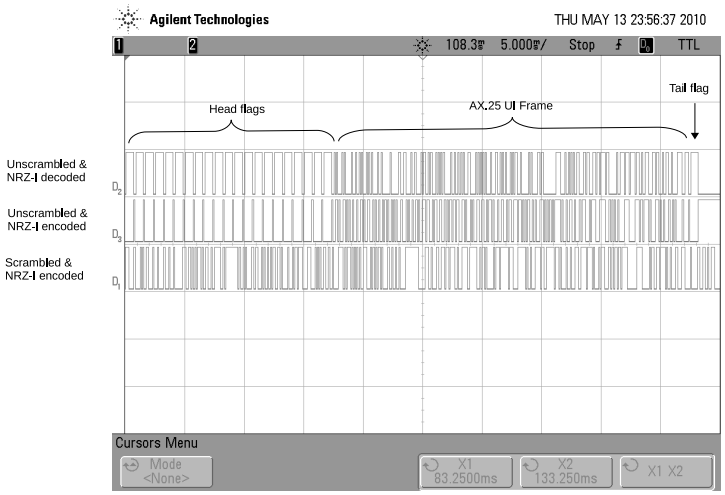


Figure 5.14: Received AX.25 frame where the raw bit stream together with an unscrambled and a NRZ-I decoded signal from the debugging lines are represented.

Chapter 6

Availability and efficiency analyzes

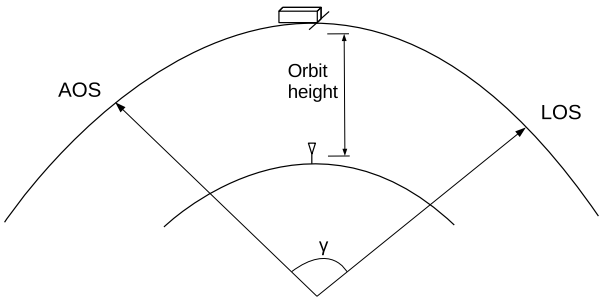


Figure 6.1: Satellite availability over a ground station.

We are describing the efficiency of the communication architecture as how effective the process is of delivering telemetry data from the satellite to the ground station. In determining the efficiency of the communication architecture, we need to define which parameters that influence the efficiency.

Calculating the throughput for informational data can give us a theoretical value of the amount of informational data that can be delivered during an available time frame.

To estimate the throughput of data we need to have a model for the availability of the data link. The availability can be defined as the total time the

data link is functional during a given time interval. The parameters that has the most decisive deterministic influence on the availability is the time window between the Acquisition of Signal (AOS), and the Loss of Signal (LOS) from the satellite.

6.1 Available time window

From the Keplerian parameters for a circular LEO in Chapter 3, we can calculate the available time window in the following way:

We are interested in the time between 0° elevation. For an orbit height of 400km, we find $\gamma = [-20 \dots 20]$. The fraction of the satellites period, will then be.

$$\frac{40}{360} \cdot 2\pi r^2$$

This leads to the calculations of the orbital period for the velocity V and the period T of the satellite:

$$V \approx \sqrt{\frac{G}{rs}}\ [km/s] \quad T = \frac{2\pi \cdot rs}{V}\ [s]$$

Where the standard gravitational parameter G is the product of the gravitational constant and the mass of the earth, and rs is the sum of the earths radius and the orbit height.

These calculations expects conditions when the satellite has a pass straight above the horizon, normally this is not the situation. The satellites orbit, expected passes and available time window can be described with equations using more parameters.

We need to define the lowest elevation angle for when communication is achievable. At 0° the surface of the earth could block transmission or introduce unwanted noise that would cause poorly reception.

6.2 Tracking the satellite

Describing the satellites orbit relative to time can be calculated by equations, and software can be used to track the position of the satellite. Common satellite orbits are available for downloading as Two Line Element (TLE) files. These can be used as input to simulation performed by software. By using one of these applications it is an easy task to simulate a known satellites orbit over time. Thus finding the availability of a link between that satellite and a fixed ground station. An example of a TLE for the NCUBE-2 satellite

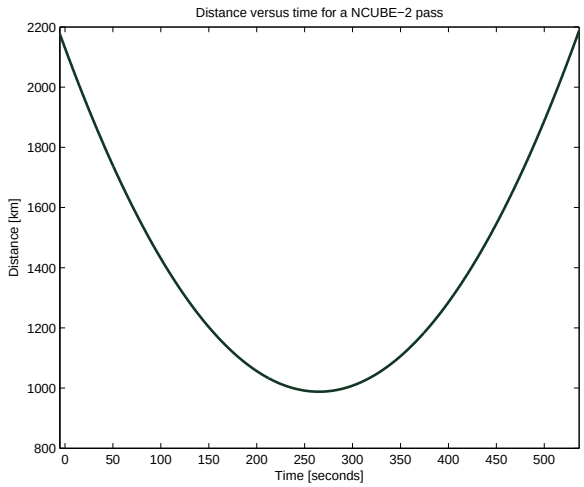


Figure 6.2: Time versus distance for NCUBE-2 passing a ground station in Oslo.

looks like this:

```
NCUBE-2
1 28897U 05043H 10031.68509664 .00000113 00000-0 32893-4 0 620
2 28897 98.0326 280.4337 0017567 357.1249 2.9862 14.60023307219362
```

6.3 Calculating the availability

Since the orbital parameters of CubeSTAR are currently unknown, the following calculations will be based on the orbital parameters of NCUBE-2. By using the TLE file from the previous section, we can predict when the satellite is in range for communication. Orbitron[1] is a application that can calculate and predict satellite orbits and availability from TLE files, I used it for getting the following results.

These predictions will show how often the satellite has a pass in range of communications, and for how long time we are able to achieve communication at the given pass. The elevation angle defines when AOS and LOS occurs. Thus setting the time for when it is possible to obtain communication and when it will be lost. The minimum elevation angle is set to 10° to filter out lower elevation passes and taking account for the drawbacks in these.

Results for one pass of NCUBE-2 is presented in Table 6.1 on the next page.

[1] More information about Orbitron:
http://www.stoff.pl

Time	Elevation angle [°]	Distance [km]
2010-04-30 10:58:00 AOS	10.1	2130
2010-04-30 11:02:25 MET	41.3	988
2010-04-30 11:06:51 AOS	10.1	2141

Table 6.1: One pass of NCUBE-2 over Oslo Ground Station.

AOS	00:23:18	02:01:40	08:24:01	09:59:48	11:38:56	21:49:57	23:25:30
LOS	00:32:35	02:07:55	08:30:55	10:09:09	11:46:17	21:52:49	23:34:40
ΔTime [s]	557	375	414	561	441	172	550

Table 6.2: Availability for NCUBE-2 over Oslo Ground Station in a 24 hour time period.

Measuring the availability over a 24-hour time period, gives the time periodes of availability presented in Table 6.2. This is measured for 2010-05-01. The total time it is possible to achieve communication for that day is 3070 seconds or approximately 51 minutes, that gives an availability of 3.6% during a 24-hour time period. This is a typical example for the availability of a LEO satellite, and will give an approximation of the availability of CubeSTAR. These results can be used to calculate the amount of telemetry data that is possible to receive in one or multiple satellite passes.

6.4 Calculating the throughput

The time it takes for one frame being sent and to it reaches the receiver, is the sum of the propagation time in Equation 6.1 and the time it takes to put the whole frame on the channel in Equation 6.2. The sum of these equations will be the time from the first bit leaves the transmitter, until the last bit is received by the receiver. Other delays related to processing and the overall communication architecture will come in addition to this time.[5]

$$\frac{\text{distance [km]}}{\text{Speed of light (300.000km/s)}} = \text{[s]} \tag{6.1}$$

$$\frac{\text{Frame Size [bits]}}{\text{Capacity [bps]}} = \text{[s]} \tag{6.2}$$

By using the calculations and results presented in this chapter together with the measured delays in Section 5.5 on page 56, we can calculate a theoretical throughput of telemetry data that can be achieved in a given time frame that could represent a single pass or multiple passes of CubeSTAR.

Table 6.3 on page 67 shows results from calculations made for a time period of 600 seconds. This represent a long pass over a ground station, and the numbers can be scaled to represent the time of an actual pass. The throughput of the true informational data has been calculated for the bit rates of 1200 bps and 9600 bps. A bit error of 0, 10^{-4} and 10^{-5} has been introduced for giving an example of how this can effect the transmission.

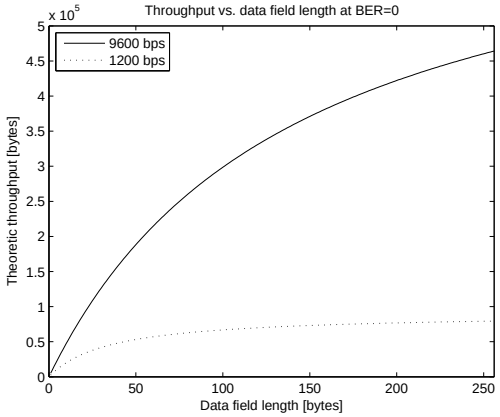
It is assumed that the BER is independent and uniformly distributed. This is an assumption that in real life would not be realistic, but makes a good example of what can be achievable. This can be representative for how the transmission over time will be effected by errors related to the BER. The Packet Error Rate (PER) can from the BER be expressed as the error probability of a symbol p_s of m length in a gaussian or binary symmetric channel with bit error probability p_b. Equation 6.3 gives this relationship, and the symbol p_s can be a data frame. [6]

$$\text{PER} = p_s = 1 - (1 - p_b)^m \tag{6.3}$$

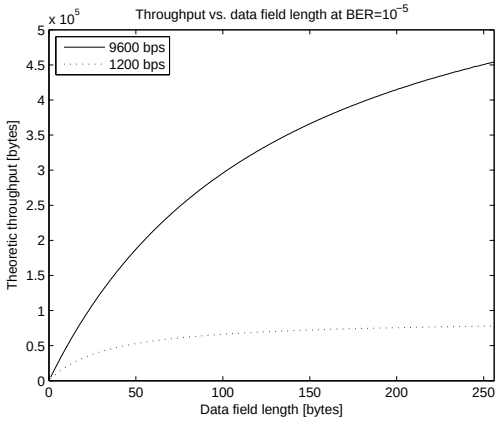
The following results was obtained by calculations done in Matlab. Pseudo code for how the calculation where made is shown in Listing 6.1. This is used for generating the graphical representation in Figure 6.3 on the next page, where the same results are represented by different utilizing of the length of the informational data in the frames. The available time period is set to 600 seconds for these calculations. For simplicity, the distance for calculating the propagation time is set to 2500 km, as it affects the overall transmission time in a minor degree.

Listing 6.1: Pseudo code in matlab for calculating the throughput.

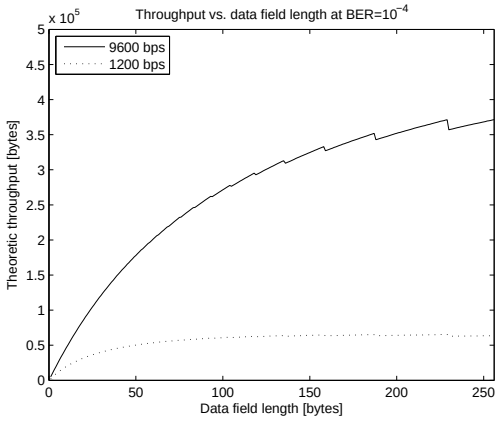
```
1  while time <= availability
2    number_of_bits = number_of_bits + total_frame_size;
3    time = time + ((preamble_bytes + total_frame_bytes ) /
         capacity); % transmit time
4    time = time + propagation_time + processing_delay;
5
6    if number_of_bits >= (1/ber)
7      lost = lost + 1;
8      number_of_bits = 0;
9    else
10     recv = recv + 1;
11   end
12
13 end
```



(a)



(b)



(c)

Figure 6.3: Theoretic throughput versus size of informational data, over a time frame of 600 seconds.

Capacity	**1200 bps**			**9600 bps**		
Bit Error Rate	**0**	**10^{-4}**	**10^{-5}**	**0**	**10^{-4}**	**10^{-5}**
Total received frames	265	212	260	1813	1451	1773
Total lost frames	0	53	5	0	362	40
Packet error rate	0%	20%	1.9%	0%	20%	2.2%
Bytes transfered	73140	58512	71760	505827	404829	494667
Protocol bytes	5565	4452	5460	43512	34824	42552
Info bytes	67575	54060	66300	462315	370005	452115

Table 6.3: Calculated throughput over a time window of 600 seconds.

6.5 Utilizing the communication architecture

The availability and throughput presented and calculated in this chapter are based on the total time frame from AOS to LOS. To utilize this in a practical implementation, the satellite would have to continuously transmit telemetry data to the ground, or the satellite immediately have to know when it is in range to achieve communication with the ground station.

There exists two different approaches in how to effectively utilize this. Since the power budget will be limited in a CubeSat, the transmitter will have to be turned off when communication with a ground station is not available. When the satellite is in range with the ground station it could deliver the telemetry data in bursts, by triggering on an uplink command or by onboard knowledge of the positional and orbital data.

6.5.1 Burst mode over ground station

Figure 6.4 on the following page presents a practical use of the available time window calculated earlier in this chapter, together with the delays that will influence the efficiency in a burst mode transmission.

From the time the satellite will have a line of sight to the ground station until the time when AOS is detected there could be delays introduced if the satellite is not in *CW mode*. This delay will be succeeded by a delay in tuning the ground station radio to the beacon signal and decoding it until the satellite enters *RX mode*. The satellite is then available for opening the link for transmitting telemetry data. This can be initiated by a command from the ground station. After processing the command at the satellite side and initiating packet transfer mode in *TX mode*, the satellite can deliver telemetry data until the time when LOS is reached.

This is the real amount of time that telemetry data can be downloaded from the satellite at a specific ground station in one pass. The delays introduced

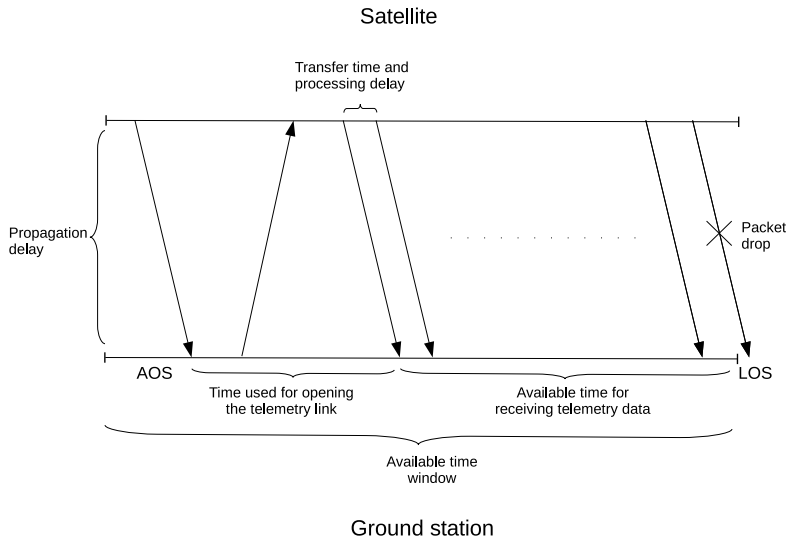


Figure 6.4: Available time window for communication represented together with the time used for opening the data link, and the achievable time for telemetry download.

during this time are the propagation time and transfer time from Equation 6.1 and 6.2 on page 64, together with the processing time in the satellite between consecutive frames. Optional retransmissions of possible lost data will influence the throughput achieved in this time frame. In this setup the satellite will retain the telemetry data to be transmitted in memory, until it receives a command from the ground station to open the data link and transmit a burst of stored telemetry data.

Actual numbers representing the time taken to open the datalink will need to be tested in a complete system.

6.5.2 Distributed ground station networks

By utilizing the fact that several CubeSat projects and amateur radio enthusiasts have ground stations around the globe, capable of communicating over the proposed protocol for CubeSTAR, we already have a distributed network of ground stations. They can be used to share resources so other CubeSat projects and radio amateurs can download telemetry data from CubeSTAR and pass it to the project over a land based internet connection. This has

already been proven to work with the moredB application used on CalPoly's CubeSats and the Rascal software for the Delfi-C3. SwissCube also offers software for downloading and decoding telemetry. The telemetry that these softwares decode, are often housekeeping data transmitted in a beacon mode. A two-way communication with the satellite have not been implemented in these decoding softwares that are distributed to other ground stations.

The above methods could be combined to a scenario where ground stations around the globe posses a software application for communicating with CubeSTAR, where uplink commands can be used to initiate a burst mode of telemetry download over the ground station. This data can then be forwarded to the CubeSTAR project. The GENSO project discussed in Section 2.2.4 on page 12 is an example of this solution.

6.6 Mission specific requirements

The requirements for the amount of data generated by the scientific mission of CubeSTAR that will need to be transmitted to ground, have not been published yet. How the above calculations meets theses mission specific requirements are based on preliminary calculations.

The scientific experiment has these preliminary requirements/modes (K. S. Jacobsen, pers. comm.):

1. Initial instrument testing. Produces 1.32 MB of data per interval.
2. Complete scientific coverage with full resolution. Produces 45.4 MB of data per orbital period.
3. Complete scientific coverage, onboard processed. Produces 1.25 MB of data per orbital period.
4. Irregularity survey mode. Produces 0.0086 MB of data per orbital period.

For the first three requirements it is easily seen that the available bandwidth is not capable of supporting the download of the specified amounts of data in one pass over a ground station. However the first three requirements are not real-time dependant, hence the measurements can be made in one orbital period and the data delivery can be distributed over several passes.

The fourth mode of 8.6 KB is well within the throughput limits of one pass, and it will be possible to run this continuously if desired.

By using an example from Table 6.3 on page 67, with a capacity of 9600 bps and an approximated BER of 10^{-4}, we find the informational throughput to be 370 KB for a period of 600 seconds. This leads to 370 KB/600 s = 616 byte per second of pure informational data. From Table 6.2 on page 64,

an available total time window of 3070 seconds was found in a 24-hour period. Again we use the availability calculations for NCUBE-2, since the orbital parameters of CubeSTAR are not known.

The total achievable throughput for this 24-hour period will then be:

$$616 \text{ byte/s} \cdot 3070 \text{ seconds} = 1893.600 \text{ KB}$$

Thus data produced from one interval in mode 1 or one orbit in mode 3 can be transmitted in less than one 24-hour period, with the above approximations and the assumption that minimum 72% of the available time window can be used for continuously telemetry. The data from one orbit in mode 2 would require at least 24 x 24-hour periods to be able to successfully receive all transmitted telemetry data.

Since these calculations are not based on the orbital parameters for CubeSTAR they will not be accurate, but they can serve as guidelines for the order of magnitude in what could be achievable. The calculations can be reused when more information about the orbit, practical usage of the communication protocol, and other requirements are set.

Chapter 7

Tests and results

In Chapter 6 a theoretic analyze of the efficiency of the communication protocol was presented. Testing the communication protocol will support the throughput calculations, and ensure that the developed source code for the communication sub-system, together with the hardware used works accordingly to the specified standards and design proposals.

7.1 Test methodologies

The test methodologies are structured as a test bench with user interaction at both the satellite side node and the ground station node.

These tests will evaluate the following performance and functionality of the implementation:

- Ensuring the proper program flow.
- Verifying proper decoding and encoding of AX.25 UI frames.
- Evaluating the performance of the communication protocol in therms of data throughput and PER.

7.2 Test applications

The test applications developed in LabView consists of an application for the satellite side node, and one for the ground station node. Together they form a framework for testing and evaluating the communication protocol between the communication sub-system and a satellite ground station.

The satellite side test application communicates with the microcontroller over an USART interface, where commands can be transmitted to and from

the test application. These commands trigger an interrupt on the microcontroller, that processes the received data immediately. It is possible to represent received AX.25 data encoded in the KISS format described in Section 5.2.1 on page 44. Figure 7.1(a) on page 74 shows a screenshot of the satellite side test application with data received from the CanSat kit.

The *Configure frame settings* box can set configuration for the amount of preamble bytes, additional head flags, delay between consecutive frames and additional tail flags. By changing these values it is possible to find the optimal settings for ensuring correctly decoding of all frames at the receiver. These settings must be configured before any transmitting is done, because it buffers the header data that is fixed for each frame.

In the *Transmit data* box, the functions for generating the different packets can be called. The informational data to send along can be specified here. Resending all frames that are in the transmit buffer can be called from the *Resend frames* button.

From the *Received data* box, AX.25 UI frames received in KISS format are represented. The received KISS data is decoded and all the decoded frame data are presented.

There is a test for measuring the throughput of informational data the system is able to transmit and receive in the *Throughput test* box. It is possible to configure how much informational data to put in the frame to transmit, and how many frames to transmit. If no number of frames is given, it will transmit until stopped. When started the frames are encoded and transmitted individually, an optional delay between the frames can be set in the *Configure frame settings* box. The info field in the frame consists of a packet identifier for the test and a sequence number, succeeded by the amount of bytes configured in the dialog box.

It is possible to monitor the amount of received informational data and PER in the satellite side test application, if frames sent from i.e. a ground station are encoded so.

The ground station side test application interfaces with a TNC, and decode KISS encoded AX.25 data. MixW is used in this setup, but the application can interface with any TNC capable of operating in KISS mode. The throughput test is evaluated and results are displayed. There is an option for transmitting data. The data is then KISS encoded and sent over the serial port it is configured for.

From the *Reception* box that is identical to the one in the satellite side test application, informational data and packet type of the last received frame can be evaluated.

The *Throughput Measurement* box presents the status and results of the

throughput test. Received and lost frames can be kept track on for finding PER. The delay between the two last frames is measured. The *Total Data received* indicates the total amount of informational data that was received since the program was started. The *Elapsed time* field indicates how long the program has run. From the *Sequence number log* box it is possible to inspect which frames where received and which where lost, this is the sequence number extracted from the packet identificator in the frame.

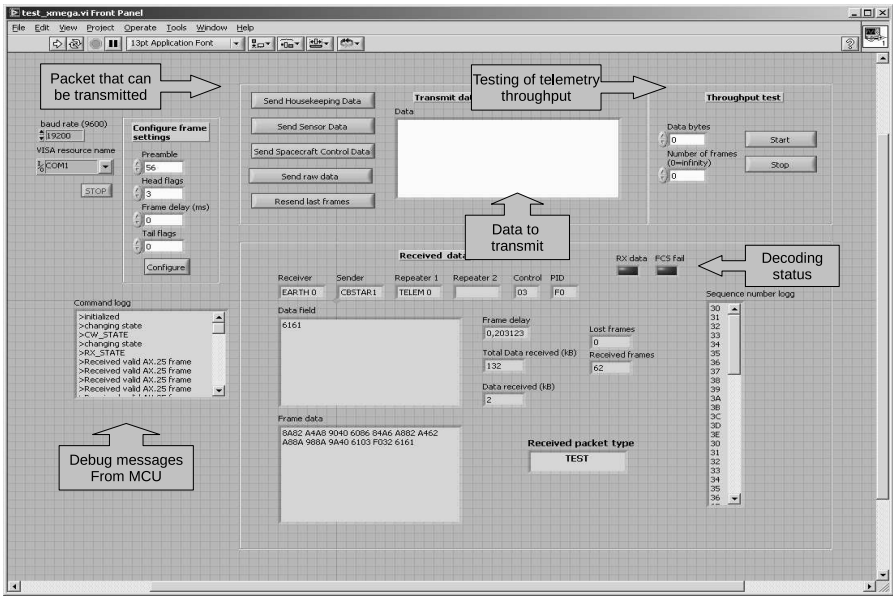
A screenshot of MixW, the software TNC can be seen in Figure 7.2 on page 75

7.3 Results

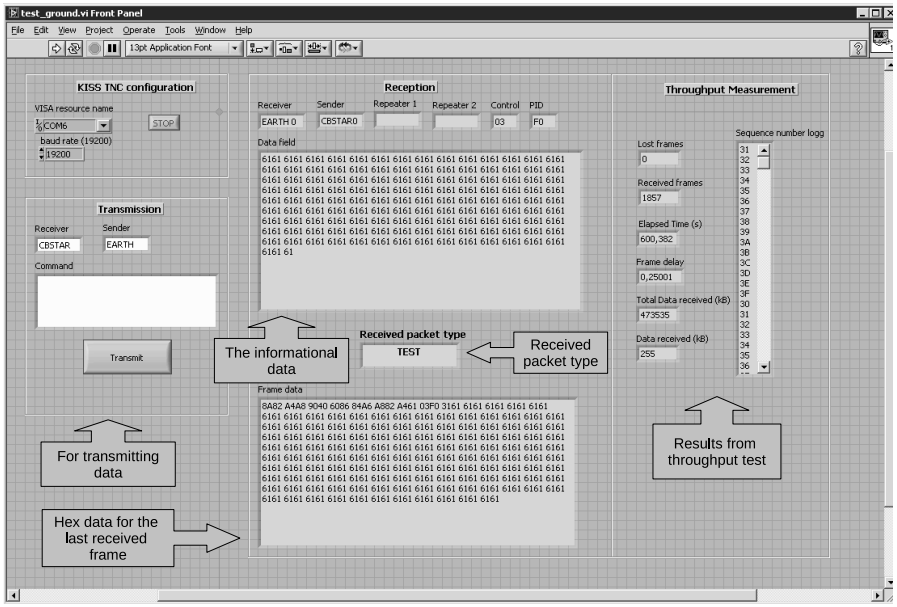
By utilizing the test methodologies with the test applications, it has been verified that the main program for the communication sub-systems performs according to the specified program flow. The system is able to decode proper AX.25 UI frames according to the specified 9600 bps standard, and the frames can be decoded by commercially available amateur radio equipment. Reception of AX.25 UI frames was successfully handled and decoded by the system. Results from a test run of both uplink an downlink is presented in the screenshots of the test application in Figure 7.1 on the following page.

The frames were in the test configured to a 56 byte preamble and three additional head flags prior to the frames. The delay was set to 0 ms, and a delay of 11.25 ms between consecutive frames was measured. This configuration shows a 77% amount of pure informational data in a transmitted frame with a fully utilized info field.

The throughput measurements reveals a total transfer of approximately 473 KB of informational data over a time period of approximately 600 seconds. This adapts well with the calculations done in Chapter 6 for the informational throughput at 9600 bps, without bit errors. Because of the link margin in the test setup, the results show no loss of frames.



(a) Test application for the satellite side, interfacing with the microcontroller.



(b) Test application for the ground station side, interfacing with MixW software TNC.

Figure 7.1: Screenshot of the test applications.

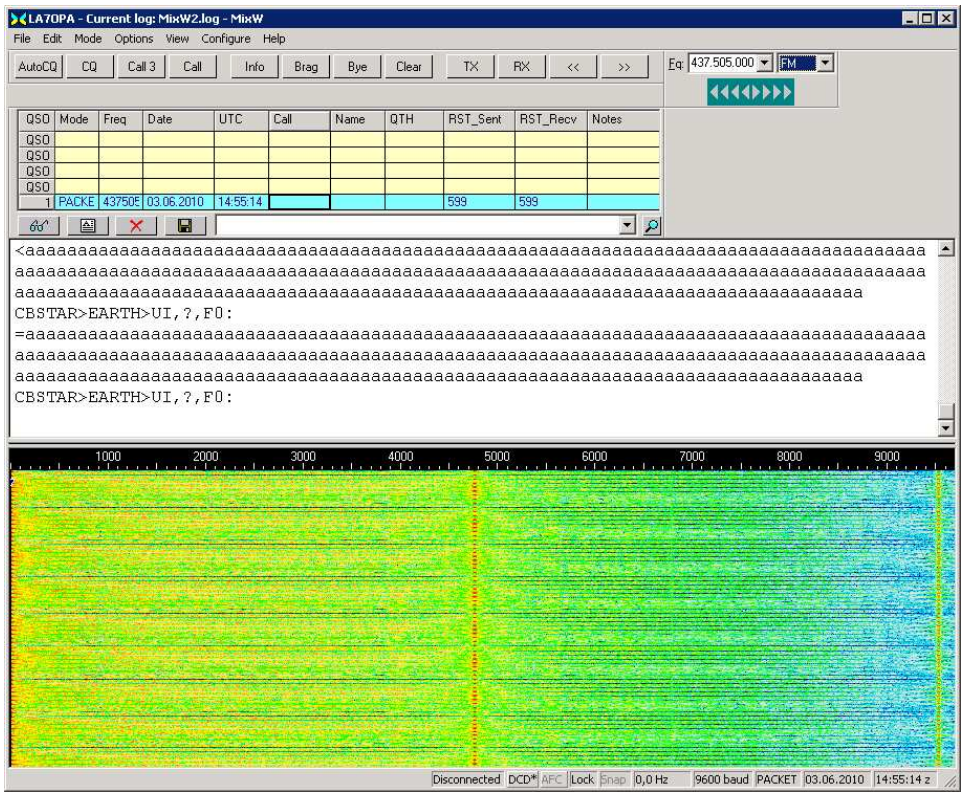


Figure 7.2: MixW, software TNC used in conjunction with the test applications. The white text box displays the received *Info*, *Address*, *PID* and *Control* data. This data is then KISS encoded before sent to the test application over the virtual serial port.

Chapter 8

Discussion

8.1 Evaluation of the implementation

Although the common used commmunication protocols on CubeSats are based on AX.25, it has not been easy to find documentation about how to actual implement it. However I found that the connection-less mode of AX.25 over UI frames could be a decant solution. The physical layer encoding techniques have been hard to find technical references for, so the process of describing these are the results of time consuming debugging and testing.

The implemented communication protocol is interoperable with commercial available amateur radio hardware. This makes it possible for others who possess the correct equipment, capable to communicate with CubeSTAR. Hence the project can utilize a distributed network of ground stations.

In this thesis the implementation of the AX.25 protocol according to the 9600 bps G3RUH standard is proposed to be used in the CubeSTAR satellite for both uplink and downlink. The selected transceiver is not capable of transmitting AFSK, and FSK with a frequency separation of 1 KHz is not possible. It will thereby not be possible to use the 1200 bps standard of AX.25. This problem was also encountered with a CC1000 transceiver in the CP2 CubeSat mission at Cal Poly[24].

A downgrade of the data rate to 1200 bps could improve the link budget results, but would limit the bandwidth. Thus shrink the data throughput. If the project would find that the link budget can not support the 9600 bps data rate, an other transceiver must be used that is capable of transmitting AFSK or FSK with a frequency separation of 1 KHz.

My implementation of the communication protocol, has potential for optimizing. As the results from the measurements in Section 5.5 states, there is a processing delay prior to each frame. AX.25 is a bit-oriented protocol

in therms that each bit will need to be evaluated to perform the bit stuffing and NRZ-I encoding. This is time consuming since the frames will need to be encoded and buffered before transmitted to the transceiver. The *FCS* calculation and scrambling is performed bit-by-bit, and can be done more effective.

Since the implementation is based on UI-frames, it is not guaranteed that all transmitted data can be received. The communication protocol provides a service of best efforts delivery. This service can be strengthened by applying more of the error handling features described in Section 2.1.1. How this can be achieved are discussed in the following sections.

The constraints of the system are set by the communication architecture, communication protocol, the hardware used, and the link budget results. Other evaluations that should be performed when the data transfer requirements for the different sub-systems, and the scientific experiment elaborates could be:

Data rate versus frequencies in use — The need for higher data rates will raise the need for using more bandwidth. Higher frequency bands as the S-band, will then have to be considered.

Power available versus link budget – The power available for the communication system to meet the expectations for the link budget must be verified. If lower power is available, then the link budget will need to be adapted accordingly, resulting in lower data rates.

Availability versus functionality – It must be ensured that adapting the protocol to be compatible with other ground stations does not set to many limitations for the communication sub-system.

8.2 Alternative protocols

This thesis has presented an implementation of a communication protocol based on AX.25 UI frames. Although this seems to be the *de facto* standard on how to implement CubeSat communication, there exists alternatives that could be evaluated if the project time frame can support this.

The FX.25 protocol

In 2006 the FX.25 protocol [25] was presented [26] as an extension to AX.25 for supporting FEC. Since AX.25 only offers an error detecting feature, received frames contaminated with a single bit error will be dropped at the receiver. This can lead to poor performance at low SNR levels.

The FX.25 protocol suggest FEC to improve performance in these situations by applying an error correcting feature. The implementation is based on redundancy added as a "wrapper" around the AX.25 frame, that is preserved in its original way. This way the protocol will be backwards compatible with already existing implementations for receiving and decoding AX.25 frames.

If FX.25 is not implemented at the receiver, it will consider the FEC redundancy as channel noise and attempt to decode the received AX.25 frame anyway, however without the error correcting feature.

The CCSDS protocol

The Consultative Committee for Space Data Systems (CCSDS) defines standards and technical recommendations for space communication, and maintains a collection of these. Various AX.25 implementations have dominated the communication protocols for CubeSats, but CCSDS based formats are being used.

Gomspace is a company founded as a spin-off from the danish CubeSat project AAU-Cubesat, they deliver a communication sub-system with a CCSDS compatible implementation of a communication protocol. This implementation supports a FEC scheme for error correction of received data. The system will need a special TNC at the ground station. [27]

If using a CCSDS based protocol over amateur radio frequencies, it must be ensured that the criterion for using these frequencies are met. Additional hardware at the ground station side is needed to support the communication protocol implementation at the satellite. Before an official standard for using CCSDS in a CubeSat exists, a variety of implementations will need to be supported at the ground station side if sharing ground station resources among CubeSat projects will be a reality.

A custom protocol

The protocol standards does not exists to make things more complex, rather they exists for making things easier. However some times it would be more convenient to define a new standard. The reason could be that the existing standards does not meet the expected requirements or services, or they are too complex to implement for the scenario they are to be used in.

By defining a new protocol it is possible to design it directly according to the rest of the communication architecture. Error controlling features and all physical layer methods can be fully adapted to the mission specific and other technical requirements.

The drawbacks in using a custom protocol is that it must be implemented at both the satellite side and ground station side. It must be ensured that both nodes can support the protocol. If additional hardware needs to be used at the ground station, the protocol is limited to work only with that ground station.

In the CC1101 transceiver there is a packet handling feature with a framing format that supports FEC. If it on the satellite side communication sub-system would be convenient to use this format, it must be implemented on the ground station side. The encoding and decoding algorithms, together with the FEC algorithms must be open and known.

A custom protocol could violate the terms of using the amateur radio band frequencies, if it is not a public available format.

8.3 Future work

As the CubeSTAR project evolves and the rest of the sub-systems are completed, they will need to be integrated with the communication sub-system to test telemetry transfer with the ground station. The first step in this process is the integration of the communication protocol implemented in this thesis, with the communication sub-system PCB and HAL. The communication protocol and HAL must be adapted to function according to the program flow presented in this thesis. A change from the "serial synchronous mode" to the "packet handling mode" in the transceiver will need to applied. In this mode the transmission and reception of correctly encoded AX.25 frames will again need to be verified.

After the integration of the communication sub-system, testing must be done to ensure correct functionality and the performance must be evaulated. The LabView test setup that I created in this project can be used to evaluate the efficiency of the communication protocol again. The preamble length and synchronization of the scrambler function could need adjustments and therefore need to be evaluated. Since the test applications are developed in LabView it is possible to connect instruments for more testing, e.g. RF measurements. Testing the system with a SNR that match the real value of the space link, will help to evaluate the PER. All this testing can be integrated to a LabView application for easier control of the input parameters and evaluation of the results. This would determine if the implementation described in this thesis performs within the requirements of BER, or if FEC will be of great advantage. A decision will then need to be taken if to pursue with this implementation, or to evaluate implementations of FX.25, CCSDS or other communication protocols.

The source code for the communication protocol will need to be optimized,

this could deliver data more effective and shrink the processing delays. This optimizing can be done in conjunction with the process of integration with the HAL and the transceiver "packet handling mode". Optimized functions can be used for calculating the *FCS* in blocks by a look up table, and the process of bit stuffing, NRZ-I encoding and scrambling can be optimized to be more effective. If other TNC types are to be used at the ground station, it will need to be inspected that the transceiver can detect the header of each frame as they are scrambled. The scrambling function should ensure a fixed pseudo-random sequence for the header that the transceiver can detect in "packet handling mode". It should be verified that the same CRC function is used on the *FCS* fields in other TNC's. As this function is not defined in the AX.25 protocol reference[15], other implementations could exist.

When the communication requirements for the rest of the satellite sub-systems and the scientific experiment are set, they must agree upon the usage of the communication protocol. This usage have I referred to the application layer of the communication architecture at the satellite side. Once the progress of the sub-systems continues, the communication requirements for the housekeeping data and other commands will need to be set. This way an efficient utilization of the communication protocol can be defined. The error handling processes will need to be evaluated, based on the results from PER tests and how critical the loss of the specified data will be. An ARQ technique for certain packet types can then be implemented.

The usage of the communication protocol with a TNC at the ground station side is described in this thesis. This can be used to develop an application to communicate with CubeSTAR. The application can be responsible for sending commands and displaying received and decoded telemetry data. By keeping track of possible lost and erroneous data, the application can initiate retransmission and acknowledge messages for the data that will need this. This is defined as the application layer of the communication architecture at the ground station side. By developing the application in a language based on free software, it is easier for other CubeSat projects to use this application at their ground station to communicate with CubeSTAR and forward received telemetry data to the project. Either the raw AX.25 frames can be TCP/IP encapsulated and forwarded, or only the data of interest can be forwarded. This architecture can utilize the distributed network of ground stations for achieving more satellite availability. Thus more received data for the project.

By using the suggested implementation of the communication protocol and developing the described application for the ground station, it is possible to have a setup that is compatible with the GENSO project and also a independent solution that can utilize distributed ground stations. A solution that is independent of GENSO differs in the way that it will need active user intervention at the ground stations where it will be used, if remote access is

not available.

8.4 Conclusion

In this thesis I have described the process of choosing a data link layer protocol to be used at CubeSTAR, and proven that the implementation of an AX.25 based communication protocol is not too complex to implement in this given time frame. This implementation of the protocol is described through the hardware used, and the software developed. I have created a library capable of encoding and decoding the link layer frames for the communication protocol, together with software using the library for the communication sub-system. The performance and constraints regarding this implementation have been showed with link budgets, efficiency analyzes, and practical tests of the communication protocol.

Since this thesis is from the first generation of masters projects in the CubeSTAR project it has been important to document the development of the implementation together with constraints, trade-offs, and alternative protocols. As the project evolves and new issues are met, this will give valuable input to the project on how the communication protocol can be modified or changed. The test applications I have created will give other students a simple user interface for testing and evaluating the communication protocol. Other tests can be integrated with these test applications.

The separated layers of the communication architecture I presented in Chapter 4 together with the separate library I created for AX.25 frames, will make the implementation adaptable for changes. The microcontroller and transceiver can be changed without big intervention in the source code.

How the implementation can be integrated with the rest of the communication sub-system and interface with the communication architecture, has been described. This should provide a good reference in how to integrate and utilize the communication protocol when other sub-systems are in place. This thesis will also be of great advantage for other projects interested in implementing and adapting an AX.25 based communication protocol.

References

[1] Tore André Bekkeng. Prototype Development of a Multi-Needle Langmuir Probe System. Master's thesis, University of Oslo, Department of Physics, 2009.

[2] K. S. Jacobsen, A. Pedersen, J. I. Moen and T. A. Bekkeng. A new Langmuir probe concept for rapid sampling of space plasma electron density. *Measurement Science and Technology*, Volume 21, 2010.

[3] Bernard Sklar. *Digital Communication*. Prentice Hall, 2001.

[4] International Telecommunications Union. *Handbook On Satellite Communications*. Wiley, Third edition, 2002.

[5] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education International, 4th edition, 2003.

[6] Jahn Lutz, Werner. *Satellite Systems for Personal and Broadband Communications*. Springer, 2000.

[7] The International Amateur Radio Union. Amateur radio satellites – information for developers of satellites planning to use frequency bands allocated to the amateur-satellite service. http://www.iaru.org/satellite/IARUSATSPEC_REV15.6.pdf (2010-07-16).

[8] The APRS Working Group. APRS Protocol Reference, 2000.

[9] R. Dean Straw, editor. *The ARRL Handbook*. ARRL – the national association for Amateur Radio, 83th edition, 2006.

[10] Walter L. Morgan and Gary D. Gordon. *Communications Satellite Handbook*. Wiley, 1989.

[11] Phil Beavis (GENSO System Engineer). A Global Educational Network for Satellite Operations. In *QB50 Workshop*, 18 th November 2009.

[12] James R. Wertz and Wiley J. Larson. *Space Mission Analysis And Design*. Space Technology Library, third edition edition, 2008.

[13] Jeremy Allnut Timothy Pratt, Charles Bostian. *Satellite Communication*. Wiley, Second Edition edition, 2002.

[14] Jim Zyren and Al Petrick. Intersil, Tutorial on Basic Link Budget Analysis. Application note
http://www.sss-mag.com/pdf/an9804.pdf (2010-05-06).

[15] *AX.25 Link Access Protocol for Amateur Packet Radio*, version 2.2 edition, July 1998.

[16] Bill Newhall. *The Cyclic Redundancy Check (CRC) for AX.25*.
http://ecee.colorado.edu/~newhallw/TechDepot/AX25CRC/CRC_for_AX25.pdf
(2009-08-04).

[17] James Miller. The 9600 bps modem. Packet–speed, more speed, and applications: A collection of advanced packet methods and activities from ARRL publications and other sources, 1995.
http://www.amsat.org/amsat/articles/g3ruh/109.html (2010-01-14).

[18] Johan Tresvig. Design of a Prototype Communication System for the CubeSTAR Nano-Satellite. Master's thesis, University of Oslo, Department of Physics, 2010.

[19] Atmel. AVR1005: Getting started with XMEGA. Application Note.

[20] Chipcon. SmartRF Studio 7 Overview (Rev. B). User Guide.

[21] Chipcon. CC1101 Low-Power Sub-1 GHz RF Transceiver (Rev. F). Datasheet.

[22] Henning Vangli. Construction of a Remotely Operated Satellite Ground Station for Low Earth Orbit Communication. Master's thesis, University of Oslo, Department of Physics, 2010.

[23] Mike Chepponis (K3MC) and Phil Karn (KA9Q). The KISS TNC: A simple Host-to-TNC communications protocol. 1987.
http://www.ka9q.net/papers/kiss.html (2010-03-20).

[24] Chris Noe. Design and Implementation of the Communications Subsystem for the Cal Poly CP2 Cubesat Project. *California Polytechnic State University*, June 11, 2004.

[25] Jim McGuire, Ivan Galysh, Kevin Doherty, Hank Heidt, Dave Neimi. FX.25 Forward Error Correction Extension to AX.25 Link Protocol For Amateur Packet Radio. Technical report, 2006.

[26] Jim McGuire. FX.25 Forward Error Correction Extension to AX.25 Link Protocol. In *The 29th Annual American Radio Relay League and Tucson Amateur Packet Radio Digital Communications Conference*, 2006.

[27] Gomspace. Nanocom U480 datasheet, 2010.

Appendix A

Link budget results

Link budget calculations are performed according to the parameters in Table A.1.

Uplink calculations are presented in Table A.2 on the next page and downlink calculations are presented in table A.3 on page 89.

Link budget parameters			**Unit**
	Distance	1432.0	km
	Elevatioon angle	10.0	°
	Bit rate	9600	bps
	Uplink frequency	433.0	MHz
	Downlink frequency	433.0	MHz
Ground station parameters			
	Latitude	59.94	°
	Longitude	10.72	°
	Height	0	km

Table A.1: Link budget parameters

Uplink parameters			
	Transmitter amplifier power	18.75	dBW
	Feeder losses	5.50	dBi
	Transmitter antenna gain	22.0	dBi
	Antenna efficiency	70.0	%
	Antenna polarisation	45.0	°
	EIRP	35.25	
Uplink losses			
	Path losses	148.29	dB
	Gazeous attenuation	0	dB
	Rain attenuation	0	dB
	Cloud attenuation	0	dB
	Scintillation losses	0	dB
	Polarisation losses	3.0	dB
	Total losses	151.29	dB
Uplink gains			
	Receiver antenna gain	0	dBi
	Feeder losses	0	dB
	LNA noise figure	1.0	dB
	Antenna temperature	120.0	K
	Spacecraft temperature	280.0	K
	LNA gain	30.0	dB
	Receiver noise temperature	290.0	K
	Receiver G/T	-27.36	dB/K
Uplink budget results			
	Link margin	15.0	dB
	$C/N0$	70.21	
	$E_b/N0$	30.38	

Table A.2: Uplink budget

Downlink parameters			
	Transmitter amplifier power	0	dBW
	Feeder losses	3.0	dBi
	Transmitter antenna gain	0	dBi
	Antenna efficiency	70.0	%
	Antenna polarisation	45.0	°
	EIRP	-3.0	
Downlink losses			
	Path losses	148.29	dB
	Gazeous attenuation	0	dB
	Rain attenuation	0	dB
	Cloud attenuation	0	dB
	Scintillation losses	0	dB
	Polarisation losses	3.0	dB
	Total losses	151.29	dB
Downlink gains			
	Receiver antenna gain	22.0	dBi
	Feeder losses	5.5	dB
	Feeder noise temperature	100.0	K
	Clear sky noise temperature	68.0	K
	LNA gain	18.0	dB
	LNA noise figure	2.04	dB
	Receiver cable loss	0.90	dB
	Receiver noise temperature	1000	K
	Receiver noise temperature	22.32	dBK
	Receiver G/T	-5.82	dB/K
Downlink budget results			
	Link margin	15.0	dB
	$C/N0$	53.49	
	$E_b/N0$	13.67	

Table A.3: Downlink budget

Appendix B

Source code listings

Listing B.1: main.c

```
/*
 * ================================================
 *
 *       Filename:  main.c
 *
 *    Description: Main file for CubeSTAR COMM system
 *
 *       Compiler:  avr-gcc
 *
 *         Author:  Markus Grønstad (markusg@ieee.org)
 *
 *          The CubeSTAR project
 *
 * ================================================
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <inttypes.h>
#include <util/delay.h>
#include <string.h>
#include <stdio.h>
#include "main.h"

/*  AX.25 UI frame info and buffers */
TX_FRAME_t tx_frame;
TX_BUFFER_t tx_buffer;
RX_BUFFER_t rx_buffer;
RX_FRAME_t rx_frame;
FRAME_CONFIGURATION_t frame_configuration;

TX_RING_BUFFER_t tx_packet_buffer;
RX_RING_BUFFER_t rx_packet_buffer;
USART_BUFFER_t usartrx_buffer;
```

```
35
36 /*  Global declarations */
37 volatile uint8_t current_state;
38 volatile uint8_t inbit;
39 volatile uint8_t usart_data;
40 volatile uint8_t frame_counting;
41 volatile uint8_t frames_to_resend;
42
43 /*  For testing only */
44 volatile uint16_t num_test_frames;
45 volatile uint8_t cont_test_frames;
46 volatile uint8_t num_test_data;
47
48
49 /** Interrupt vectors  **/
50 /*  Interrupt on leading edge from transceiver clock, handle
       received bit in synchronus receive mode. */
51 ISR(PORTE_INT0_vect)
52 {
53   inbit = DATA_PORT.IN & DATA_PIN;
54   add_bit(inbit, &rx_frame, &rx_buffer, SCRAMBLE);
55 }
56
57 /*  Interrupts when receiving data from USART */
58 ISR(USARTC0_RXC_vect)
59 {
60   usart_data = USART.DATA;
61
62   if(usartrx_buffer.ptr < USART_SIZE)
63   {
64     from_usart();
65   }
66 }
67
68 /* Clears the USART receive buffer */
69 void usart_buffer_clr()
70 {
71   uint8_t i;
72   for(i=0; i<255; i++)
73   {
74     usartrx_buffer.buffer[i] = 0x00;
75     usartrx_buffer.ptr = 0;
76   }
77 }
78
79 /*  Handle data from the USART */
80 /*  Used to trigger packet sending and tests */
81 /*  Replace with functions for SPI bus */
82 void from_usart()
83 {
84   usartrx_buffer.buffer[usartrx_buffer.ptr++] = usart_data;
85
86   switch(usartrx_buffer.buffer[0])
87   {
```

```
88      /*  Create COMMAND frame */
89      case 0x63:  // c
90        if(usartrx_buffer.ptr > 0)
91        {
92          if(usartrx_buffer.ptr == (uint16_t) (usartrx_buffer.
                buffer[1] + 2))
93          {
94            create_frame(CMD, (usartrx_buffer.buffer + 2),
                  usartrx_buffer.buffer[1]);
95            usart_buffer_clr();
96            set_state(TX_STATE);
97          }
98        }
99        break;
100
101       /*  Create HOUSEKEEPING frame */
102     case 0x68:  // h
103       if(usartrx_buffer.ptr > 0)
104       {
105         if(usartrx_buffer.ptr == ((uint16_t) (usartrx_buffer.
                buffer[1] + 2)))
106         {
107           create_frame(HK, (usartrx_buffer.buffer + 2),
                  usartrx_buffer.buffer[1]);
108           usart_buffer_clr();
109           set_state(TX_STATE);
110         }
111       }
112       break;
113
114       /*  Create PAYLOAD frame */
115     case 0x70:  // p
116       if(usartrx_buffer.ptr > 0)
117       {
118         if(usartrx_buffer.ptr == (uint16_t) (usartrx_buffer.
                buffer[1] + 2))
119         {
120           create_frame(PAYLOAD, (usartrx_buffer.buffer + 2),
                  usartrx_buffer.buffer[1]);
121           usart_buffer_clr();
122           set_state(TX_STATE);
123         }
124       }
125       break;
126
127     case 0x77:  // w
128       if(usartrx_buffer.ptr > 0)
129       {
130         if(usartrx_buffer.ptr == (uint16_t) (usartrx_buffer.
                buffer[1] + 2))
131         {
132           create_frame(RAW, (usartrx_buffer.buffer + 2),
                  usartrx_buffer.buffer[1]);
133           usart_buffer_clr();
```

```
134             set_state(TX_STATE);
135           }
136         }
137         break;
138
139         /*  Set frame configuration */
140       case 0x74:  // t
141         if(usartrx_buffer.ptr == 5)
142         {
143           config_frame(usartrx_buffer.buffer[1], usartrx_buffer.
                buffer[2], usartrx_buffer.buffer[3], usartrx_buffer
                .buffer[4]);
144           usart_buffer_clr();
145         }
146         break;
147
148         /*  Throughput test */
149       case 0x31:  // 1
150         if(usartrx_buffer.ptr == 4)
151         {
152           // sett data til test rammene, antall eller uendelig
                sending
153           test_1(usartrx_buffer.buffer[1], usartrx_buffer.buffer
                [2], usartrx_buffer.buffer[3]);
154           usart_buffer_clr();
155           set_state(TX_TEST_STATE);
156         }
157         break;
158
159         /*  Resend last n frames */
160       case 0x72:  // r
161         frames_to_resend = tx_packet_buffer.buffer_fill;
162         usart_buffer_clr();
163         set_state(TX_STATE);
164         break;
165
166         /*  Stop sending */
167       case 0x73:
168         usart_buffer_clr();
169         num_test_frames = 0;
170         cont_test_frames = FALSE;
171         set_state(CW_STATE);
172         break;
173
174
175       default:
176         usart_buffer_clr();
177         break;
178     }
179 }
180
181 /*  Initialises TX/RX USART */
182 /*  RX used to initiate test functions and send frames */
183 /*  TX used to send debug messages and received KISS data */
```

```
184 void init_usart()
185 {
186   /*  USART rx */
187   USART_PORT.DIRCLR = PIN2_bm;
188   USART_PORT.DIRSET = PIN3_bm;
189
190   //USART.BAUDCTRLA = 103;  /*  9600 bps @ 6 MHz clock */
191   USART.BAUDCTRLA = 51; /*  19200 bps @ 6 MHz clock */
192   USART.BAUDCTRLB = 0;
193
194   USART.CTRLC |= USART_CHSIZE_8BIT_gc |
          USART_CMODE_ASYNCHRONOUS_gc | USART_PMODE_DISABLED_gc;
195
196   USART.CTRLA |= USART_RXCINTLVL0_bm;
197   PMIC.CTRL |= PMIC_LOLVLEN_bm;
198   USART.CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
199
200   usartrx_buffer.ptr = 0;
201 }
202
203 /*  Configures TX standard parameters for all frames */
204 void config_frame(uint8_t preamble, uint8_t head, uint8_t tail,
        uint8_t delay)
205 {
206   uint8_t i;
207
208   init_tx_buffer(&tx_buffer);
209   init_tx_frame(&tx_frame);
210
211   frame_configuration.delay = delay;      /*  Delay between
          frames */
212   frame_configuration.preamble = preamble;  /*  Bytes of
          preamble */
213   frame_configuration.add_head = head;    /*  Number of
          additional head flags */
214   frame_configuration.add_tail = tail;    /*  Number of
          additional tail flags */
215
216
217   /*  Buffer preamble and header data */
218   for(i=0; i<frame_configuration.preamble; i++)
219     buffer_byte(0x00, &tx_buffer, FALSE);
220
221   for(i=0; i<frame_configuration.add_head; i++)
222     buffer_byte(0x7E, &tx_buffer, SCRAMBLE);
223
224   set_address(&tx_frame, RECEIVER, 0x00, SENDER, 0x00, NULL, 0
          x00, NULL, 0x00);
225   buffer_header(&tx_frame, &tx_buffer, SCRAMBLE);
226
227
228 #ifdef DEBUG
229   send_debug((uint8_t*) ">frame configured\0");
230 #endif
```

```
231 }
232
233 /*  Test function #1: Throughput test */
234 void test_1(uint8_t data, uint8_t frames_lo, uint8_t frames_hi)
235 {
236   uint16_t frames;
237   frames = (frames_hi << 8) | frames_lo;
238
239   if(frames > 0)
240     num_test_frames = frames;
241   else
242     cont_test_frames = TRUE;
243
244   num_test_data = data;
245 }
246
247 /*  Creates an packet, ready to be AX.25 endoded. Put packet in
        TX ring buffer. */
248 void create_frame(enum commands id, uint8_t *data, uint8_t size
      )
249 {
250   uint8_t data_field[256];
251   uint8_t i;
252   i = 0;
253
254   switch(id)
255   {
256     case TEST:
257       for(i=0; i<size; i++)
258         data_field[i+1] = data[i];
259       data_field[0] = (tx_frame.frame_counter & 0x0F) | id;
260       if(tx_frame.frame_counter++ == 0x0F)
261         tx_frame.frame_counter = 0;
262
263       break;
264
265     case CMD:
266       for(i=0; i<size; i++)
267         data_field[i+1] = data[i];
268       data_field[0] = id;
269       break;
270
271     case HK:
272       for(i=0; i<size; i++)
273         data_field[i+1] = data[i];
274       data_field[0] = id;
275       break;
276
277     case PAYLOAD:
278       for(i=0; i<size; i++)
279         data_field[i+1] = data[i];
280       data_field[0] = (tx_frame.frame_counter & 0x0F) | id;
281       if(tx_frame.frame_counter++ == 0x0F)
282         tx_frame.frame_counter = 0;
```

```
283       break;
284
285     case RAW:
286       for(i=0; i<size; i++)
287         data_field[i] = data[i];
288       break;
289   }
290
291   data_field[i] = '\0';
292
293   tx_ring_buffer_push(&tx_packet_buffer, data_field);
294 }
295
296 /* Reads data to be sent from TX ring buffer. Encodes data, and
       sends it. */
297 void send_frame()
298 {
299 #ifdef DEBUG
300   send_debug((uint8_t*) ">sending frame\0");
301 #endif
302   uint8_t buf_data[256], i;
303   uint16_t end;
304   tx_ring_buffer_pop(&tx_packet_buffer, buf_data);
305   //if(tx_ring_buffer_pop(&tx_packet_buffer, buf_data))   /*
        Can be used for casting exception when ring buffer is
        full */
306   //  return;
307
308   for(end=0; end<=255; end++)
309   {
310     if(buf_data[end] == '\0')
311       break;
312   }
313   end--;
314
315   reset_tx_frame(&tx_frame);
316   clear_tx_buffer(&tx_buffer);
317
318   set_info(&tx_frame, buf_data, end);
319
320   buffer_frame(&tx_frame, &tx_buffer, SCRAMBLE);
321
322   for(i=0; i<frame_configuration.add_tail; i++)
323     buffer_byte(0x7E, &tx_buffer, SCRAMBLE);
324
325   bang_out(&tx_buffer);
326   _delay_ms(frame_configuration.delay);
327 }
328
329 /*  Clock out buffered frame in serial synchronus transmit mode
       */
330 void bang_out(TX_BUFFER_t *buf)
331 {
332   uint16_t i, j;
```

```
333   uint8_t bmask, bit;
334
335   for(i=0; i<buf->index; i++)
336   {
337     byte_out(buf->frame_buffer[i]);
338   }
339
340   bmask = 0x01;
341   /*  Clock out only valid bits in the last byte */
342   for(j=0; j<buf->bit_counter; j++)
343   {
344     bit = ((buf->frame_buffer[i]) & bmask) >> j;
345     bit_out(bit);
346     bmask <<= 1;
347   }
348
349   /*  Set ouput to low  */
350   bit_out(0x00);
351 }
352
353 /*  Clock out buffered frame in serial synchronus transmit mode
      , byte by byte */
354 void byte_out(uint8_t byte)
355 {
356   uint8_t bit;
357   uint8_t i;
358   uint8_t bmask;
359
360   bmask = 0x01;
361
362   for(i=0; i< 8; i++)
363   {
364     bit = (byte & bmask) >> i;
365     bit_out(bit);
366     bmask <<= 1;
367   }
368 }
369
370 /*  Clock out buffered frame in serial synchronus transmit mode
      , bit by bit*/
371 void bit_out(uint8_t bit)
372 {
373   /*  Wait for signal from transceiver */
374   while(!(CLK_PORT.IN & CLK_PIN));
375   DATA_PORT.OUT = bit;
376   while(CLK_PORT.IN & CLK_PIN);
377 }
378
379 /*  Received valid AX.25 frame. Handle data in KISS format. */
380 /*  Function can also be changed to read AX.25 fields directly
      through RX_FRAME_t struct, e.g. check for
381  *  valid addresses */
382 void handle_data(RX_BUFFER_t *buf, RX_FRAME_t *frame)
383 {
```

```
384   cli();
385   uint8_t i, byte;
386 
387 #ifdef DEBUG
388   send_debug((uint8_t*) ">Received valid AX.25 frame\0");
389 #endif
390 
391   while(!((USART.STATUS & USART_DREIF_bm) !=0));
392   USART.DATA = 0xC0;
393   while(!((USART.STATUS & USART_DREIF_bm) !=0));
394   USART.DATA = 0x00;
395 
396   for(i=0; i<(buf->data_bytes)-2; i++)
397   {
398     byte = buf->data_buffer[i];
399 
400     if(byte == 0xDB)
401     {
402       while(!((USART.STATUS & USART_DREIF_bm) !=0));
403       USART.DATA = 0xDB;
404       while(!((USART.STATUS & USART_DREIF_bm) !=0));
405       USART.DATA = 0xDD;
406     }
407     else if(byte == 0xC0)
408     {
409       while(!((USART.STATUS & USART_DREIF_bm) !=0));
410       USART.DATA = 0xDB;
411       while(!((USART.STATUS & USART_DREIF_bm) !=0));
412       USART.DATA = 0xDC;
413     }
414     else
415     {
416       while(!((USART.STATUS & USART_DREIF_bm) !=0));
417       USART.DATA = byte;
418     }
419   }
420 
421   while(!((USART.STATUS & USART_DREIF_bm) !=0));
422   USART.DATA = 0xC0;
423 
424   init_rx_buffer(buf);
425   init_rx_frame(frame);
426 
427   sei();
428 }
429 
430 
431 /*  Evaluate received data in RX ring buffer */
432 void handle_rx_buffer(RX_RING_BUFFER_t *rx_rb)
433 {
434   uint8_t retval, byte, bit, bmask;
435   uint8_t rx_data[RX_RB_SIZE];
436   uint16_t i;
437 
```

```
438   do
439   {
440     retval = rx_ring_buffer_pop(&rx_packet_buffer, rx_data);
441     for(i=0; i<RX_RB_SIZE; i++)
442     {
443       byte = rx_data[i];
444       bmask = 0x01;
445       for(i=0; i<8; i++)
446       {
447         bit = (byte & bmask) >> i;
448         bmask <<= 1;
449         add_bit(bit, &rx_frame, &rx_buffer, SCRAMBLE);
450       }
451     }
452   }
453   while(!retval);
454 }
455
456 /**  Ring buffer functions **/
457 /*  TX ring buffer used to store packets prior to AX.25
      encoding and sending */
458 void tx_ring_buffer_init(TX_RING_BUFFER_t *tx_rb)
459 {
460   tx_rb->count = 0;
461   tx_rb->buffer_fill = 0;
462   tx_rb->head = &tx_rb->buffer[0];
463   tx_rb->tail = &tx_rb->buffer[0];
464 }
465
466 /*  Push tx_data in to tx_rb buffer */
467 uint8_t tx_ring_buffer_push(TX_RING_BUFFER_t *tx_rb, const
      uint8_t *tx_data)
468 {
469   uint16_t i;
470
471   if(tx_rb->count == RB_SIZE)
472     return 1;
473
474   for(i=0; i<TX_RB_SIZE; i++)
475     *(tx_rb->head)++ = tx_data[i];
476
477   tx_rb->count++;
478   if(tx_rb->buffer_fill < RB_SIZE)
479     tx_rb->buffer_fill++;
480
481   if(tx_rb->head == &tx_rb->buffer[TX_RB_SIZE * RB_SIZE])
482     tx_rb->head = &tx_rb->buffer[0];
483
484   return 0;
485 }
486
487 /*  Pop tx_data from tx_rb buffer */
488 uint8_t tx_ring_buffer_pop(TX_RING_BUFFER_t *tx_rb, uint8_t *
      tx_data)
```

```
489 {
490   uint16_t i;
491 
492   //  if(tx_rb->count == 0)
493   //    return 1;
494 
495   for(i=0; i<TX_RB_SIZE; i++)
496     tx_data[i] = *(tx_rb->tail)++;
497 
498   tx_rb->count--;
499 
500   if(tx_rb->tail == &tx_rb->buffer[TX_RB_SIZE * RB_SIZE])
501     tx_rb->tail = &tx_rb->buffer[0];
502 
503   return 0;
504 }
505 
506 /*  Moves tail pointer to the oldest data, for resending the
       whole buffer */
507 void tx_ring_buffer_reset(TX_RING_BUFFER_t *tx_rb)
508 {
509   tx_rb->temp = tx_rb->tail;
510   tx_rb->count += frames_to_resend;
511 
512   if(tx_rb->buffer_fill < RB_SIZE)
513     tx_rb->tail = &tx_rb->buffer[0];
514   else
515     tx_rb->tail = tx_rb->head + TX_RB_SIZE;
516 }
517 
518 /*  Set tail pointer to the correct place */
519 void tx_ring_buffer_set(TX_RING_BUFFER_t *tx_rb)
520 {
521   tx_rb->tail = tx_rb->temp;
522 }
523 
524 /*  RX ring buffer used for storing data from transceiver */
525 void rx_ring_buffer_init(RX_RING_BUFFER_t *rx_rb)
526 {
527   rx_rb->count = 0;
528   rx_rb->head = &rx_rb->buffer[0];
529   rx_rb->tail = &rx_rb->buffer[0];
530 }
531 
532 /*  Pushes rx_data to rx_rb buffer. Called from low-level
       firmware when using SPI bus */
533 uint8_t rx_ring_buffer_push(RX_RING_BUFFER_t *rx_rb, uint8_t *
       rx_data)
534 {
535   uint16_t i;
536 
537   if(rx_rb->count == RB_SIZE)
538     return 1;
539 
```

```
540   for(i=0; i<RX_RB_SIZE; i++)
541     *(rx_rb->head)++ = rx_data[i];
542
543   rx_rb->count++;
544
545   if(rx_rb->head == &rx_rb->buffer[RX_RB_SIZE * RB_SIZE])
546     rx_rb->head = &rx_rb->buffer[0];
547
548   return 0;
549 }
550
551 /*  Get rx_data from rx_rb buffer.  */
552 uint8_t rx_ring_buffer_pop(RX_RING_BUFFER_t *rx_rb, uint8_t *
       rx_data)
553 {
554   uint16_t i;
555
556   if(rx_rb->count == 0)
557     return 1;
558
559   for(i=0; i<RX_RB_SIZE; i++)
560     rx_data[i] = *(rx_rb->tail)++;
561
562   rx_rb->count--;
563
564   if(rx_rb->tail == &rx_rb->buffer[RX_RB_SIZE * RB_SIZE])
565     rx_rb->tail = &rx_rb->buffer[0];
566
567   return 0;
568 }
569
570 /* Enable external clock as clock source */
571 void init_clk()
572 {
573   OSC.XOSCCTRL = OSC_FRQRANGE_2TO9_gc | OSC_XOSCSEL_EXTCLK_gc;
574   OSC.CTRL |= OSC_XOSCEN_bm;
575   while ( (OSC.STATUS & OSC_XOSCRDY_bm) == 0);
576   CCP=0xD8;
577   CLK.CTRL = CLK_SCLKSEL_XOSC_gc;
578   OSC.CTRL &= ~OSC_RC2MEN_bm;
579 }
580
581 /*  Do all initialization */
582 void initialize()
583 {
584   init_clk();
585   init_rx_buffer(&rx_buffer);
586   init_rx_frame(&rx_frame);
587   init_usart();
588
589   CLK_PORT.DIRCLR = CLK_PIN;  /*  Clock from transceiver */
590
591 #ifdef DEBUG
592   DEBUG_PORT1.DIRSET = PIN0_bm; /*  Debug in decoding received
```

```
        frames */
593   DEBUG_PORT2.DIRSET = PIN0_bm;
594 #endif
595
596   rx_ring_buffer_init(&rx_packet_buffer);
597   tx_ring_buffer_init(&tx_packet_buffer);
598
599   /* Initialize global variables */
600   frame_counting = FALSE;
601   cont_test_frames = FALSE;
602   num_test_data = 0;
603   num_test_data = 0;
604   frames_to_resend = 0;
605
606 #ifdef DEBUG
607   send_debug((uint8_t*) ">initialized\0");
608 #endif
609 }
610
611 /*  Change the state of the COMM system */
612 void set_state(uint8_t next_state)
613 {
614   cli();
615 #ifdef DEBUG
616   send_debug((uint8_t*) ">changing state\0");
617 #endif
618
619   switch(next_state)
620   {
621     case RX_STATE:
622       DATA_PORT.DIRCLR = DATA_PIN;
623       CLK_PORT.INTCTRL = PMIC_LOLVLEN_bm;
624       CLK_PORT.INT0MASK = CLK_PIN;
625       CLK_PORT.PIN2CTRL = PORT_ISC_RISING_gc;
626 #ifdef DEBUG
627       send_debug((uint8_t*) ">RX_STATE\0");
628 #endif
629
630       break;
631
632     case TX_STATE:
633       DATA_PORT.DIRSET = DATA_PIN;
634       CLK_PORT.INTCTRL &= ~PMIC_LOLVLEN_bm;
635       CLK_PORT.INT0MASK &= ~CLK_PIN;
636       CLK_PORT.PIN2CTRL &= ~PORT_ISC_RISING_gc;
637 #ifdef DEBUG
638       send_debug((uint8_t*) ">TX_STATE\0");
639 #endif
640
641       break;
642
643     case TX_TEST_STATE:
644       DATA_PORT.DIRSET = DATA_PIN;
645       CLK_PORT.INTCTRL &= ~PMIC_LOLVLEN_bm;
```

```
646       CLK_PORT.INT0MASK &= ~CLK_PIN;
647       CLK_PORT.PIN2CTRL &= ~PORT_ISC_RISING_gc;
648 #ifdef DEBUG
649       send_debug((uint8_t*) ">TX_TEST_STATE\0");
650 #endif
651 
652       break;
653 
654     case CW_STATE:
655 #ifdef DEBUG
656       send_debug((uint8_t*) ">CW_STATE\0");
657 #endif
658 
659       break;
660   }
661 
662   current_state = next_state;
663 
664   sei();
665 }
666 
667 
668 /*  Main function */
669 int main()
670 {
671   uint8_t data_buf[255];
672   initialize();
673 
674   PMIC.CTRL = PMIC_LOLVLEN_bm;
675 
676   set_state(CW_STATE);
677 
678   sei();
679 
680   while(1)
681   {
682 
683     /*  State machine for the communication system. Check
          continously which state we are in, and perform
684      *  functions upon current state we are in */
685     switch(current_state)
686     {
687       case TX_STATE:
688         // calls low level functions for transceiver settings
689         if(frames_to_resend > 0)
690         {
691           tx_ring_buffer_reset(&tx_packet_buffer);
692           while(frames_to_resend > 0)
693           {
694             send_frame(data_buf);
695             frames_to_resend--;
696           }
697           tx_ring_buffer_set(&tx_packet_buffer);
698           break;
```

```
699
700         }
701
702         while(tx_packet_buffer.count > 0)
703           send_frame(data_buf);
704
705         set_state(CW_STATE);
706         break;
707
708       case RX_STATE:
709         // calls low level functions for transceiver settings
710         if(rx_frame.valid)
711           handle_data(&rx_buffer, &rx_frame);
712
713         break;
714
715       case CW_STATE:
716         // calls low level functions for transceiver settings
717         // enables CW beacon mode
718         // can be called from a timer interrupt
719         set_state(RX_STATE);
720         break;
721
722     }
723
724     /*  State for transmitting test data, can be removed */
725     if(current_state == TX_TEST_STATE)
726     {
727       uint8_t data[num_test_data];
728       uint8_t i;
729       for(i=0; i<num_test_data; i++)
730         data[i] = 'a';
731
732       if(cont_test_frames)
733       {
734         while(cont_test_frames)
735         {
736           create_frame(TEST, data, num_test_data);
737           send_frame();
738         }
739       }
740       else if(num_test_frames > 0)
741       {
742         while(num_test_frames-- > 0)
743         {
744           create_frame(TEST, data, num_test_data);
745           send_frame();
746         }
747       }
748       set_state(CW_STATE);
749     }
750   }
751
752   return 0;
```

```
753 }
```

Listing B.2: main.h

```
1  /*
2   * ================================================
3   *
4   *       Filename:  main.h
5   *
6   *    Description: Main header file for CubeSTAR COMMsystem
7   *
8   *       Compiler:  avr-gcc
9   *
10  *         Author:  Markus Grønstad (markusg@ieee.org)
11  *
12  *          The CubeSTAR project
13  *
14  * ================================================
15  */
16
17 #ifndef MAIN_H
18 #define MAIN_H
19
20 #include "ax25.h"
21 #ifdef DEBUG
22 #include "debug.h"
23 #endif
24
25 #define TRUE 1
26 #define FALSE 0
27
28 /*  Define ports */
29 #define CLK_PORT PORTE
30 #define DATA_PORT PORTE
31 #define CLK_PIN PIN2_bm
32 #define DATA_PIN PIN0_bm
33 #define USART_PORT PORTC
34 #define USART USARTC0
35
36
37 /*  States for the communication system */
38 enum states
39 {
40   CW_STATE,
41   RX_STATE,
42   TX_STATE,
43   TX_TEST_STATE
44 };
45
46 /*  G3RUH scramble */
47 #define SCRAMBLE TRUE
48
49 /*  Addresses, 6 ascii charaters, pad with white spaces */
```

```
50 #define SENDER "CBSTAR"
51 #define RECEIVER "EARTH "
52
53 /*  Size of ring buffer for packets */
54 /*  Must be of 2^n size */
55 #define RB_SIZE 8
56 #define TX_RB_SIZE 256
57 #define RX_RB_SIZE 256
58
59 #define USART_SIZE 258
60
61 /*  Data buffer for receiving USART data */
62 typedef struct USART_BUFFER
63 {
64   uint8_t buffer[USART_SIZE];
65   uint16_t ptr;
66 }USART_BUFFER_t;
67
68 /*  Ring buffer for buffering several packets prior to sending
       */
69 typedef struct TX_RING_BUFFER
70 {
71   uint8_t buffer[TX_RB_SIZE * RB_SIZE];
72   uint8_t count;
73   uint8_t buffer_fill;
74   uint8_t *head;
75   uint8_t *tail;
76   uint8_t *temp;
77 }TX_RING_BUFFER_t;
78
79 /*  Ring buffer for buffering several received frames */
80 typedef struct RX_RING_BUFFER
81 {
82   uint8_t buffer[RX_RB_SIZE * RB_SIZE];
83   uint8_t count;
84   uint8_t *head;
85   uint8_t *tail;
86 }RX_RING_BUFFER_t;
87
88 /* Register for keeping frame configuration settings */
89 typedef struct FRAME_CONFIGURATION
90 {
91   uint8_t delay;
92   uint8_t preamble;
93   uint8_t add_head;
94   uint8_t add_tail;
95 }FRAME_CONFIGURATION_t;
96
97 /* Protocol identifiers for the CubeSTAR protocol */
98 /* Can be extended to other packet types here, e.g. ACK/NACK
       */
99 enum commands
100 {
101   CMD = 0x10,     // command frames
```

```
102   HK = 0x20,       // housekeeping frames
103   TEST = 0x30,     // test frames
104   PAYLOAD = 0x40,   // payload data frames
105   RAW = 0x50       // raw ascii frames
106 };
107
108 /*  Function declarations */
109 void initialize();
110
111 void add_byte(RX_BUFFER_t *buf);
112 void handle_rx_buffer(RX_RING_BUFFER_t *rx_rb);
113 void handle_data(RX_BUFFER_t *buf, RX_FRAME_t *frame);
114
115 void init_clk();
116 void init_usart();
117 void config_frame(uint8_t preamble, uint8_t head, uint8_t tail,
        uint8_t delay);
118 void create_frame(enum commands id, uint8_t *data, uint8_t size
       );
119 void send_frame();
120 void resend_frames();
121 void bang_out(TX_BUFFER_t *buf);
122 void byte_out(uint8_t byte);
123 void bit_out(uint8_t bit);
124
125 void tx_ring_buffer_init(TX_RING_BUFFER_t *tx_rb);
126 uint8_t tx_ring_buffer_push(TX_RING_BUFFER_t *tx_rb, const
       uint8_t *tx_data);
127 uint8_t tx_ring_buffer_pop(TX_RING_BUFFER_t *tx_rb, uint8_t *
       tx_data);
128 void tx_ring_buffer_reset(TX_RING_BUFFER_t *tx_rb);
129 void tx_ring_buffer_set(TX_RING_BUFFER_t *tx_rb);
130 void rx_ring_buffer_init(RX_RING_BUFFER_t *rx_rb);
131 uint8_t rx_ring_buffer_push(RX_RING_BUFFER_t *rx_rb, uint8_t *
       rx_data);
132 uint8_t rx_ring_buffer_pop(RX_RING_BUFFER_t *rx_rb, uint8_t *
       rx_data);
133
134 void usart_buffer_clr();
135 void from_usart();
136
137 void test_frame(uint8_t preamble, uint8_t head, uint8_t tail,
       uint8_t delay, uint8_t data, uint8_t frames_lo, uint8_t
       frames_hi);
138 void test_frame_command(enum commands id, uint8_t data_length);
139 void test_1(uint8_t data, uint8_t frames_lo, uint8_t frames_hi)
       ;
140 void set_state(uint8_t next_state);
141
142 #endif
```

Listing B.3: debug.c

```
/*
 * ================================================
 *
 *       Filename:  debug.c
 *
 *    Description:  Functions for debugging the
 *                  CubeSTAR COMM system
 *
 *       Compiler:  avr-gcc
 *
 *         Author:  Markus Grønstad (markusg@ieee.org)
 *
 *          The CubeSTAR project
 *
 * ================================================
 */

#include "debug.h"

/*  Send debug data over USART */
void send_debug(uint8_t *msg)
{
  uint8_t i = 0;

  if(msg[0] != 0xCB)
  {
    while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
    DEBUG_USART.DATA = 0xCA;
    while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
    DEBUG_USART.DATA = 0x00;

    while(msg[i] != '\0')
    {
      while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
      DEBUG_USART.DATA = *msg++;
    }

    while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
    DEBUG_USART.DATA = 0x0D;    // transmit CR
    while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
    DEBUG_USART.DATA = 0x0A;    // transmit CR
    while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
    DEBUG_USART.DATA = 0xCA;

  }
  else
  {
    while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
    DEBUG_USART.DATA = msg[0];
    while(!((DEBUG_USART.STATUS & USART_DREIF_bm) !=0));
    DEBUG_USART.DATA = msg[1];
  }
}
```

Listing B.4: debug.c

```
/*
 * ======================================================
 *
 *       Filename:  debug.h
 *
 *    Description:  Functions for debugging the
 *                  CubeSTAR COMM system
 *
 *       Compiler:  avr-gcc
 *
 *         Author:  Markus Grønstad (markusg@ieee.org)
 *
 *          The CubeSTAR project
 *
 * ======================================================
 */


#ifndef DEBUG_H
#define DEBUG_H

#include <avr/io.h>
#include <inttypes.h>

/*  Debug ports */
#define DEBUG_PORT1 PORTF
#define DEBUG_PORT2 PORTD

/*  Debug USART */
#define DEBUG_USART USARTC0

/*  Function declarations */
void send_debug(uint8_t *msg);

#endif
```

Listing B.5: ax25.c

```
/*
 * ======================================================
 *
 *       Filename:  ax25.c
 *
 *    Description: Library for encoding and decoding
 *             of AX.25 frames
 *
 *       Compiler:  avr-gcc
 *
 *         Author:  Markus Grønstad (markusg@ieee.org)
 *
 *          The CubeSTAR project
```

```
14  *
15  * ================================================
16  */
17
18 #include "ax25.h"
19
20 #ifdef DEBUG
21 #include "debug.h"
22 #endif
23
24 /* Bit wise calculate the FCS */
25 uint16_t calcFCS(uint8_t byte, uint16_t fcs)
26 {
27   uint8_t bit;
28
29   for(bit=0; bit<8; bit++)
30   {
31     fcs ^= (byte & 0x01);
32     fcs = (fcs & 0x01) ? (fcs >> 1) ^ 0x8408 : (fcs >> 1);
33     byte = byte >> 1;
34   }
35
36   return fcs;
37 }
38
39 /* ****************************************************
40  * Functions for Tx.
41  * ****************************************************/
42
43 /*  Initialize one AX.25 UI frame, resets all values */
44 void init_tx_frame(TX_FRAME_t *frame)
45 {
46   reset_tx_frame(frame);
47   _init_tx_frame(frame);
48 }
49
50 /*  Resets only the frame counter of an AX.25 UI frame */
51 void _init_tx_frame(TX_FRAME_t *frame)
52 {
53   frame->flag = 0x7E;
54   frame->control = 0x03;
55   frame->pid = 0xF0;
56   frame->head_fcs = 0xFFFF;
57
58   uint16_t i;
59
60   for(i=0; i<ADDR_L; i++)
61   {
62     frame->address[i] = 0;
63   }
64   frame->address_bytes = 0;
65   frame->address_count = 0;
66   frame->frame_counter = 0;
67 }
```

```
68
69 /*  Resets all values, but frame counter and address in an AX
       .25 frame */
70 void reset_tx_frame(TX_FRAME_t *frame)
71 {
72   uint16_t i;
73
74   frame->fcs = 0xFFFF;
75   for(i=0; i<INFO_L; i++)
76   {
77     frame->info[i] = 0;
78   }
79
80   frame->info_bytes = 0;
81 }
82
83 /*  Initialize one buffer for AX.25 UI frame(s) */
84 void init_tx_buffer(TX_BUFFER_t *buf)
85 {
86   uint16_t i;
87   for(i=0; i<FRAME_L; i++)
88   {
89     buf->frame_buffer[i] = 0x00;
90   }
91
92   buf->hi_count = 0;
93   buf->last_bit = 0;
94   buf->index = 0;
95   buf->bit_counter = 0;
96   buf->byte = 0x00;
97   buf->lfsr = 0L;
98   buf->_hi_count = 0;
99   buf->_last_bit = 0;
100  buf->_index = 0;
101  buf->_bit_counter = 0;
102  buf->_byte = 0x00;
103  buf->_lfsr = 0L;
104 }
105
106 /*  Set the send/recv address and the optional repeater
       addresses */
107 void set_address(TX_FRAME_t *frame, char *destination, uint8_t
       ssid1, char *source, uint8_t ssid2, char *rpt1, uint8_t
       ssid3 , char *rpt2, uint8_t ssid4)
108 {
109   uint8_t i, j;
110
111   uint8_t source_addr[7];
112   uint8_t dest_addr[7];
113   uint8_t rpt1_addr[7];
114   uint8_t rpt2_addr[7];
115
116   /*  Shift addresses one bit source the left. AX.25 standard
         */
```

```
117   for(i=0; i<6; i++)
118   {
119     dest_addr[i] = destination[i] << 1;
120     source_addr[i] = source[i] << 1;
121   }
122
123   dest_addr[6] = (ssid1 << 1) | 0x60;
124   source_addr[6] = (ssid2 << 1) | 0x60;
125   frame->address_bytes = 14;
126
127   if(rpt1)
128   {
129     frame->address_count++;
130     for(i=0; i<6; i++)
131     {
132       rpt1_addr[i] = rpt1[i] << 1;
133       frame->address_bytes++;
134     }
135     rpt1_addr[6] = (ssid3 << 1) | 0x60;
136     frame->address_bytes++;
137   }
138
139   if(rpt2)
140   {
141     frame->address_count++;
142     for(i=0; i<6; i++)
143     {
144       rpt2_addr[i] = rpt2[i] << 1;
145       frame->address_bytes++;
146     }
147     rpt2_addr[6] = (ssid3 << 1) | 0x61;
148     frame->address_bytes++;
149   }
150
151   /*  Set correct address extension bit */
152   if(!rpt1)
153     source_addr[6] |= 0x01;
154   else if((rpt1) && (!rpt2))
155     rpt1_addr[6] |= 0x01;
156
157   /*  Copy addresses to frame */
158   j = 0;
159   for(i=0; i<7; i++)
160   {
161     frame->address[j] = dest_addr[i];
162     j++;
163
164   }
165
166   for(i=0; i<7; i++)
167   {
168     frame->address[j] = source_addr[i];
169     j++;
170
```

```
171   }
172
173   if(ADDR_L > 14)
174   {
175     if(rpt1)
176     {
177       for(i=0; i<7; i++)
178       {
179         frame->address[j] = rpt1_addr[i];
180         j++;
181
182       }
183     }
184
185   }
186
187   if(ADDR_L > 21)
188   {
189     if(rpt2)
190     {
191       for(i=0; i<7; i++)
192       {
193         frame->address[j] = rpt2_addr[i];
194         j++;
195
196       }
197     }
198
199   }
200
201   /*  Pre-calculate the FCS for the header fields */
202   set_head_FCS(frame);
203 }
204
205
206 /* Set the info field and calculate FCS for the frame */
207 void set_info(TX_FRAME_t *frame, uint8_t *info, uint16_t sz)
208 {
209   uint16_t i;
210   for(i=0; i<=sz; i++)
211     frame->info[i] = info[i];
212   frame->info_bytes = sz+1;
213
214   set_infoFCS(frame);
215 }
216
217 /*  Calculates the FCS for the info field, FCS for header
      fields
218  *  must have been precalculated. I.e. address must have been
       set prior
219  *  to setting the info. */
220 void set_infoFCS(TX_FRAME_t *frame)
221 {
222   uint16_t sum, i;
```

```
223   uint8_t c;
224
225   sum = frame->head_fcs;
226
227   for(i=0; i<frame->info_bytes; i++)
228   {
229     c = frame->info[i];
230     sum  = calcFCS(c, sum);
231   }
232
233   frame->fcs = ~sum;
234 }
235
236 /* Precalculate FCS for address, controll and pid, called from
       setAddress() */
237 void set_head_FCS(TX_FRAME_t *frame)
238 {
239   int i;
240   //unsigned int sum;
241   uint16_t sum;
242
243   for(i=0; i<frame->address_bytes; i++)
244   {
245     sum = calcFCS(frame->address[i], frame->head_fcs);
246
247     frame->head_fcs = sum;
248   }
249
250   sum = calcFCS(frame->control, frame->head_fcs);
251
252   frame->head_fcs = sum;
253   sum = calcFCS(frame->pid, frame->head_fcs);
254
255   frame->head_fcs = sum;
256 }
257
258
259 /*  Inspect byte bit for bit for bitstuffing, NRZ-I encoding
       and G3RUH scrambling if necessary */
260 /*  Returns 1 if buffer is full */
261 void byte2buf(uint8_t byte, uint8_t bitstuff,  TX_BUFFER_t *buf
       , uint8_t scramble)
262 {
263   uint16_t i;
264   uint8_t bmask, bit;
265   bmask = 0x01;
266   uint8_t hi_count;
267   uint8_t outbit;
268   uint8_t lfsr_hi, lfsr_lo, lfsr_bit, scr_bit;
269
270   /*  Bitwise inspect the given byte */
271   for(i=0; i<8; i++)
272   {
273
```

```
274     bit = (bmask & byte);
275     bmask <<= 1;
276     bit >>= i;
277
278     /*  Perform NRZ-I encoding */
279     /*  Check if this bit is 1 */
280     if(bit)
281     {
282       hi_count = buf->hi_count;
283       outbit = buf->last_bit; /*  Next bit is same as previous
              */
284       buf->last_bit = outbit;
285
286       /*  Perform scrambling according to G3RUH standard */
287       if(scramble)
288       {
289         lfsr_hi = (buf->lfsr & 0x00000001L);
290         lfsr_lo = ((buf->lfsr >> 5) & 0x01L);
291         lfsr_bit = lfsr_hi ^ lfsr_lo;
292
293         scr_bit = outbit ^ lfsr_bit;
294         buf->byte |= scr_bit << buf->bit_counter;
295         buf->lfsr >>= 1;
296         if(scr_bit) buf->lfsr |= 0x00010000L;
297       }
298       else
299         buf->byte |= (outbit << buf->bit_counter);
300
301       buf->bit_counter++;
302       buf->frame_buffer[buf->index] = buf->byte;
303       buf->hi_count++;
304
305       /* Check if we have one byte to add to buffer */
306       if(buf->bit_counter == 8)
307       {
308         buf->index++;
309         buf->byte = 0x00;
310         buf->bit_counter = 0;
311       }
312
313
314       /*  Check for bit stuffing */
315       if(buf->hi_count == 5 && bitstuff == TRUE)
316       {
317         /* Perform NRZ-I encoding */
318         outbit = (~(buf->last_bit) & 0x01); /*  Next bit is
                inverse of previous */
319         buf->last_bit = outbit;
320
321         /*  Perform scrambling according to G3RUH standard */
322         if(scramble)
323         {
324           lfsr_hi = (buf->lfsr & 0x00000001L);
325           lfsr_lo = ((buf->lfsr >> 5) & 0x01L);
```

```
326           lfsr_bit = lfsr_hi ^ lfsr_lo;
327
328           scr_bit = outbit ^ lfsr_bit;
329           buf->byte |= (scr_bit << buf->bit_counter);
330           buf->lfsr >>= 1;
331           if(scr_bit) buf->lfsr |= 0x00010000L;
332         }
333         else
334           buf->byte |= (outbit << buf->bit_counter);
335
336         buf->bit_counter++;
337         buf->frame_buffer[buf->index] = buf->byte;
338         buf->hi_count = 0;
339
340         /* Check if we have one byte to add to buffer */
341         if(buf->bit_counter == 8)
342         {
343           buf->index++;
344           buf->byte = 0x00;
345           buf->bit_counter = 0;
346         }
347       }
348     }
349     /*  Check if this bit is 0 */
350     else
351     {
352       /* Perform NRZ-I encoding */
353       outbit = (~(buf->last_bit) & 0x01); /*  Next bit is
              inverse of previous */
354       buf->last_bit = outbit;
355
356       /*  Perform scrambling according to G3RUH standard */
357       if(scramble)
358       {
359         lfsr_hi = (buf->lfsr & 0x00000001L);
360         lfsr_lo = ((buf->lfsr >> 5) & 0x01L);
361         lfsr_bit = lfsr_hi ^ lfsr_lo;
362
363         scr_bit = outbit ^ lfsr_bit;
364         buf->byte |= (scr_bit << buf->bit_counter);
365         buf->lfsr >>= 1;
366         if(scr_bit) buf->lfsr |= 0x00010000L;
367       }
368       else
369         buf->byte |= (outbit << buf->bit_counter);
370
371       buf->bit_counter++;
372       buf->frame_buffer[buf->index] = buf->byte;
373       buf->hi_count = 0;
374
375       /* Check if we have one byte to add to buffer */
376       if(buf->bit_counter == 8)
377       {
378         buf->index++;
```

```
379         buf->byte = 0x00;
380         buf->bit_counter = 0;
381       }
382     }
383
384
385   }
386 }
387
388 /*  Buffer all header data and retain FCS and LFSR valus, since
389  *  this is the same for all frames.  */
390 void buffer_header(TX_FRAME_t *frame, TX_BUFFER_t *buf, uint8_t
        scramble)
391 {
392   uint8_t c, i;
393
394   /*  Buffer header flag */
395   byte2buf(frame->flag, FALSE, buf, scramble);
396
397   /*  Buffer address field */
398   for(i=0; i<frame->address_bytes; i++)
399   {
400     c = frame->address[i];
401     byte2buf(c, TRUE, buf, scramble);
402   }
403
404   /*  Buffer control field */
405   byte2buf(frame->control, TRUE, buf, scramble);
406
407   /*  Buffer pid field */
408   byte2buf(frame->pid, TRUE, buf, scramble);
409
410   buf->_hi_count = buf->hi_count;
411   buf->_last_bit = buf->last_bit;
412   buf->_index = buf->index;
413   buf->_bit_counter = buf->bit_counter;
414   buf->_byte = buf->byte;
415   buf->_lfsr = buf->lfsr;
416 }
417
418 /*  Send UI frame to buffer, and perform bit stuffing and NRZ-I
        encoding. */
419 void buffer_frame(TX_FRAME_t *frame, TX_BUFFER_t *buf, uint8_t
       scramble)
420 {
421   uint8_t c, fcs_hi, fcs_lo;
422   uint16_t j;
423
424   for(j=0; j<frame->info_bytes; j++)
425   {
426     c = frame->info[j];
427     byte2buf(c, TRUE, buf, scramble);
428   }
429
```

```
430   /*  Buffer FCS field */
431   fcs_hi = frame->fcs & 0x00FF;
432   fcs_lo = (frame->fcs & 0xFF00) >> 8;
433   byte2buf(fcs_hi, TRUE, buf, scramble);
434   byte2buf(fcs_lo, TRUE, buf, scramble);
435
436   /*  Buffer tail flag */
437   byte2buf(frame->flag, FALSE, buf, scramble);
438 }
439
440 /*  Clear all data from tx_buffer, but preamble and header data
        */
441 void clear_tx_buffer(TX_BUFFER_t *buf)
442 {
443   uint16_t i;
444   for(i=(buf->_index); i<FRAME_L; i++)
445   {
446     buf->frame_buffer[i] = 0;
447   }
448
449   buf->hi_count = buf->_hi_count;
450   buf->last_bit = buf->_last_bit;
451   buf->index = buf->_index;
452   buf->bit_counter = buf->_bit_counter;
453   buf->byte =  buf->_byte;
454   buf->lfsr =  buf->_lfsr;
455 }
456
457 /* This function can be used to buffer addidtional bytes
       outside the
458  * AX.25 frame, as preamble and additional flags. Optional
        G3RUH scrambling
459  * can also be performed. The LFSR will be updated
460  * accordingly.
461  * NRZI-encoding will be performed, but not bit stuffing.
462  */
463 void buffer_byte(uint8_t byte, TX_BUFFER_t *buf, uint8_t
       scramble)
464 {
465   uint8_t i, out_byte, bit, bmask, outbit, scr_bit;
466   uint8_t lfsr_hi, lfsr_lo, lfsr_bit;
467
468   bmask = 0x01;
469   out_byte = 0x00;
470
471   for(i=0; i<8; i++)
472   {
473     bit = (byte & bmask) >> i;
474     bmask <<= 1;
475
476     if(bit)
477       outbit = buf->last_bit; /*  Next bit is same as previous
             */
478     else
```

```
479       outbit = (~(buf->last_bit) & 0x01); /*  Next bit is
              inverse of previous */
480
481     buf->last_bit = outbit;
482
483     /*  G3RUH scramble */
484     if(scramble)
485     {
486       lfsr_hi = (buf->lfsr & 0x00000001L);
487       lfsr_lo = ((buf->lfsr >> 5) & 0x01L);
488       lfsr_bit = lfsr_hi ^ lfsr_lo;
489       scr_bit = outbit ^ lfsr_bit;
490       buf->byte |= (scr_bit << buf->bit_counter);
491       buf->lfsr >>= 1;
492       if(scr_bit) buf->lfsr |= 0x00010000L;
493     }
494     else
495       buf->byte |= (outbit << buf->bit_counter);
496
497     buf->frame_buffer[buf->index] = buf->byte;
498
499     buf->bit_counter++;
500
501     /* Check if we have one byte to add to buffer */
502     if(buf->bit_counter == 8)
503     {
504       buf->index++;
505       buf->byte = 0x00;
506       buf->bit_counter = 0;
507     }
508   }
509 }
510
511 /* *******************************************************
512  * Functions for Rx.
513  * *******************************************************/
514
515 /*  Initialize RX frame */
516 void init_rx_frame(RX_FRAME_t *frame)
517 {
518   uint16_t i;
519
520   for(i=0; i<ADDR_L; i++)
521   {
522     frame->address[i] = 0x00;
523   }
524
525   frame->address_count = 1;
526   frame->control = 0x00;
527   frame->pid = 0x00;
528
529   for(i=0; i<INFO_L; i++)
530   {
531     frame->info[i] = 0x00;
```

```
532   }
533
534   frame->info_bytes = 0;
535   frame->fcs = 0xFFFF;
536   frame->valid = FALSE;
537 }
538
539 /*  Initialize RX buffer */
540 void init_rx_buffer(RX_BUFFER_t *buf)
541 {
542   uint16_t i;
543
544   buf->data_bytes = 0;
545
546   for(i=0; i<MAX_PACKET; i++)
547   {
548     buf->data_buffer[i] = 0x00;
549   }
550
551   clear_rx_buffer(buf);
552 }
553
554 /*  Clears all meta data used when decoding, retains the
       decoded data */
555 void clear_rx_buffer(RX_BUFFER_t *buf)
556 {
557   buf->byte = 0x00;
558   buf->bmask = 0;
559   buf->flag_detected = FALSE;
560   buf->data_detected = FALSE;
561   buf->last_bit = 0x00;
562   buf->lfsr = 0L;
563   buf->hi_count = 0;
564   buf->flag_count = 0;
565 }
566
567 /*  Decodes received data bit for bit. Can be called from an
       interrupt routine or by functions that evaluate the
568  *  received data bit by bit */
569 void add_bit(uint8_t bit, RX_FRAME_t *frame, RX_BUFFER_t *buf,
       uint8_t scramble)
570 {
571   uint8_t lfsr_hi, lfsr_lo, lfsr_bit, scr_bit, cur_bit;
572   // for debug data to usart
573 #ifdef DEBUG
574   uint8_t status_led[2];
575   status_led[0] = 0xCB;
576 #endif
577
578   /*  Descramble */
579   if(scramble)
580   {
581     lfsr_hi = (buf->lfsr & 0x00000001L);
582     lfsr_lo = ((buf->lfsr >> 5) & 0x01L);
```

```
583     lfsr_bit = lfsr_hi ^ lfsr_lo;
584     scr_bit = bit ^ lfsr_bit;
585
586     buf->lfsr >>= 1;
587     if(bit)
588       buf->lfsr |= 0x00010000L;
589   }
590   else
591     scr_bit = bit;
592
593   /*  NRZ-I decode */
594   if(scr_bit)
595   {
596     cur_bit = buf->last_bit;
597     // the unscrambled data can be read here
598 #ifdef DEBUG
599     DEBUG_PORT1.OUT = 0x01;
600 #endif
601
602   }
603   else
604   {
605     cur_bit = (~(buf->last_bit)) & 0x01;
606 #ifdef DEBUG
607     // the unscrambled data can be read here
608     DEBUG_PORT1.OUT = 0x00;
609 #endif
610   }
611
612   // the nrz-i decoded data can be read here
613 #ifdef DEBUG
614   DEBUG_PORT2.OUT = cur_bit & 0x01;
615 #endif
616
617   /*  Remove bitstuffing if necessary */
618   if((buf->data_detected) && (buf->hi_count == 5) && (cur_bit
        == 0))
619   {
620     buf->hi_count = 0;
621     buf->last_bit = scr_bit;
622     return;
623   }
624
625   /*  Add decoded bit to LSB of MSB of byte. Since LSB is
        transmitted first, the decoded bit will be LSB
626    *  after 8 received bits */
627   if(cur_bit)
628   {
629     buf->byte |= 0x80;
630     buf->hi_count++;
631   }
632   else
633   {
634     buf->byte &= 0x7F;
```

```
635     buf->hi_count = 0;
636   }
637
638   buf->last_bit = scr_bit;
639
640   /* An AX.25 flag is detected, handle received data, byte wise
          */
641   if(buf->flag_detected)
642   {
643     buf->bmask++;
644     if(buf->bmask == 8)
645     {
646       buf->bmask = 0;
647
648       /*  Preamble detected, look for head flag again. */
649       if(buf->byte == 0xFF)
650         buf->flag_detected = FALSE;
651
652       /*  Consecutive AX.25 flags, dont process */
653       /*  First data byte detected */
654       if(buf->byte != 0x7E)
655       {
656         buf->data_detected = TRUE;
657       }
658       else
659       {
660         /*  Tail flag detected. Handle packet if above minimum
                packet length and under maximum length.
661          *  Can filter out som possible bogus data */
662         buf->data_detected = FALSE;
663         if((buf->data_bytes > MIN_PACKET) && (buf->data_bytes <
               MAX_PACKET))
664         {
665 #ifdef DEBUG
666           status_led[1] = 0xCE;
667           send_debug(status_led);
668 #endif
669           /*  Call for frame decoding */
670           if(decode_frame(frame, buf))
671           {
672             frame->valid = TRUE;
673             clear_rx_buffer(buf);
674             return;
675           }
676           else
677           {
678             frame->valid = FALSE;
679 #ifdef DEBUG
680             status_led[1] = 0xCD;
681             send_debug(status_led);
682 #endif
683           }
684           init_rx_buffer(buf);
685           init_rx_frame(frame);
```

```
686         }
687         else
688         {
689           buf->flag_detected = FALSE;
690           buf->data_detected = FALSE;
691           buf->data_bytes = 0;
692 
693         }
694       }
695 
696       /*  Add received data bytes to buffer */
697       if(buf->data_detected)
698       {
699         buf->data_buffer[buf->data_bytes] = buf->byte;
700         buf->data_bytes++;
701       }
702     }
703   }
704 
705   /*  Received AX.25 flag. Prepare for bytewise data handling
        */
706   if((buf->byte == 0x7E) && (buf->flag_detected == FALSE))
707   {
708     buf->flag_detected = TRUE;
709 
710   }
711 
712   /*  Make space for new bit */
713   buf->byte >>= 1;
714 }
715 
716 /*  Decode received buffer into an AX.25 UI frame */
717 uint8_t decode_frame(RX_FRAME_t *rx_frame, RX_BUFFER_t *rx_buf)
718 {
719   if(verify_frame(rx_buf))
720   {
721     uint8_t i, j, more_addresses, valid;
722 
723     more_addresses = TRUE;
724     valid = FALSE;
725     i = 0;
726     j = 0;
727 
728     /*  Find addresses */
729     while(more_addresses)
730     {
731       for(i=0; i<6; i++)
732       {
733         rx_frame->address[j] = (rx_buf->data_buffer[j] >> 1);
734         j++;
735       }
736       rx_frame->address[j] = ((rx_buf->data_buffer[j]) & 0x1E)
            >> 1;
737       if(rx_buf->data_buffer[j++] & 0x01)
```

```
738         more_addresses = FALSE;
739       else
740         rx_frame->address_count++;
741     }
742
743     /*  Find control and pid info */
744     rx_frame->control = rx_buf->data_buffer[j++];
745     rx_frame->pid = rx_buf->data_buffer[j++];
746
747     /*  Strict check for if received frame is UI frame */
748     if((rx_frame->control) == 0x03 && (rx_frame->pid = 0xF0))
749       valid = TRUE;
750
751     i = 0;
752     for(j=j; j<((rx_buf->data_bytes)-2); j++)
753     {
754       rx_frame->info[i++] = rx_buf->data_buffer[j];
755       rx_frame->info_bytes++;
756     }
757
758     rx_frame->fcs = rx_buf->data_buffer[j++];
759     rx_frame->fcs |= ((rx_buf->data_buffer[j]) << 8);
760
761     return TRUE;
762   }
763   else
764     return FALSE;
765 }
766
767 /*  Verifiy the integrity of received frame */
768 uint8_t verify_frame(RX_BUFFER_t *rx_buf)
769 {
770   uint16_t sum, i;
771   uint8_t fcs_lo, fcs_hi, _fcs_lo, _fcs_hi;
772
773   sum = 0xFFFF;
774
775   /*  Calculate and verify the FCS */
776   for(i=0; i<((rx_buf->data_bytes)-2); i++)
777   {
778     sum = calcFCS(rx_buf->data_buffer[i], sum);
779   }
780
781   fcs_hi = rx_buf->data_buffer[i++];
782   fcs_lo = rx_buf->data_buffer[i];
783
784   sum ^= 0xFFFF;
785   _fcs_hi = sum & 0x00FF;
786   _fcs_lo = (sum & 0xFF00) >> 8;
787
788   if((fcs_hi == _fcs_hi) && (fcs_lo == _fcs_lo))
789     return TRUE;
790   else
791     return FALSE;
```

```
792 }
```

Listing B.6: ax25.h

```
1  /*
2   * ================================================
3   *
4   *       Filename:  ax25.h
5   *
6   *    Description: Library for encoding and decoding
7   *                 of AX.25 frames
8   *
9   *       Compiler:  avr-gcc
10  *
11  *         Author:  Markus Grønstad (markusg@ieee.org)
12  *
13  *          The CubeSTAR project
14  *
15  * ================================================
16  */
17
18 #ifndef AX25_H
19 #define AX25_H
20
21 #ifdef DEBUG
22 #include "debug.h"
23 #endif
24
25 #include <inttypes.h>
26
27 #define MAX_PACKET 276    /*  Maximum packet size accepted,
       this includes protocol data and
28                 *  data payload. */
29 #define MIN_PACKET 21   /*  Minimum packet size accepted  */
30
31 #define ADDR_L 14     /*  Size of address field, either
       14,21,28 */
32 #define FRAME_L 400 /*  Buffer for AX.25 UI frame, extra space
       for bit stuffing, preamble and addidtional flags
33                  ,needs to be big enough to take all data.  */
34 #define INFO_L 256      /*  Allowed size of Info field, max 256
        bytes Length  */
35
36 #define TRUE 1
37 #define FALSE 0
38
39 /*  Struct for building an AX.25 UI frame */
40 /*  Stores all the data for the unencoded frame, together with
       the
41  *  precalculated FCS for the header fields. */
42 typedef struct TX_FRAME_struct
43 {
44   uint8_t flag;
```

```
45   uint8_t address[ADDR_L];
46   uint8_t control;
47   uint8_t pid;
48   uint8_t info[INFO_L];
49   uint16_t fcs;
50
51   uint16_t head_fcs;
52   uint8_t address_count;
53   uint8_t address_bytes;
54   uint16_t info_bytes;
55   uint8_t frame_counter;
56 } TX_FRAME_t;
57
58 /*  Struct for keeping data for the decoded AX.25 frame */
59 typedef struct RX_FRAME_struct
60 {
61   uint8_t address[ADDR_L];
62   uint8_t address_count;
63   uint8_t control;
64   uint8_t pid;
65   uint8_t info[INFO_L];
66   uint8_t info_bytes;
67   uint16_t fcs;
68   uint8_t valid;
69 } RX_FRAME_t;
70
71 /*  Struct for building an AX.25 UI frame */
72 /*  Stores the AX.25 encoded data of one frame */
73 typedef struct TX_BUFFER_struct
74 {
75     uint8_t frame_buffer[FRAME_L];
76     uint8_t hi_count;
77     uint8_t last_bit;
78     uint16_t index;
79     uint8_t bit_counter;
80     uint8_t byte;
81     uint32_t lfsr;
82   uint8_t _hi_count;
83   uint8_t _last_bit;
84   uint16_t _index;
85   uint8_t _bit_counter;
86   uint8_t _byte;
87   uint32_t _lfsr;
88 } TX_BUFFER_t;
89
90 /* Buffer for retaining received unencoded data */
91 typedef struct RX_BUFFER_struct
92 {
93   uint8_t byte;
94   uint8_t bmask;
95   uint8_t flag_detected;
96   uint8_t data_detected;
97   uint16_t data_bytes;
98   uint8_t data_buffer[MAX_PACKET];
```

```
 99   uint8_t last_bit;
100   uint32_t lfsr;
101   uint8_t hi_count;
102   uint8_t flag_count;
103 }RX_BUFFER_t;
104
105 /*  Function declarations */
106 uint16_t calcFCS(uint8_t c, uint16_t fcs);
107
108 /*  Tx specific function declarations */
109 void init_tx_frame(TX_FRAME_t *frame);
110 void _init_tx_frame(TX_FRAME_t *frame);
111 void reset_tx_frame(TX_FRAME_t *frame);
112 void init_tx_buffer(TX_BUFFER_t *buf);
113 void init_tx_frame(TX_FRAME_t *frame);
114 void init_tx_buffer(TX_BUFFER_t *buf);
115 void set_address(TX_FRAME_t *frame, char *destination, uint8_t
       ssid1, char *source, uint8_t ssid2, char *rpt1, uint8_t
       ssid3 , char *rpt2, uint8_t ssid4);
116 void set_info(TX_FRAME_t *frame, uint8_t *info, uint16_t sz);
117 void set_infoFCS(TX_FRAME_t *frame);
118 void set_head_FCS(TX_FRAME_t *frame);
119 void byte2buf(uint8_t byte, uint8_t bitstuff,  TX_BUFFER_t *buf
       , uint8_t scramble);
120 void buffer_header(TX_FRAME_t *frame, TX_BUFFER_t *buf, uint8_t
        scramble);
121 void buffer_frame(TX_FRAME_t *frame, TX_BUFFER_t *buf, uint8_t
       scramble);
122 void clear_tx_buffer(TX_BUFFER_t *buf);
123 void buffer_byte(uint8_t byte, TX_BUFFER_t *buf, uint8_t
       scramble);
124
125 /*  Rx specific function declarations */
126 void init_rx_frame(RX_FRAME_t *frame);
127 void init_rx_buffer(RX_BUFFER_t *buf);
128 void clear_rx_buffer(RX_BUFFER_t *buf);
129 void add_bit(uint8_t bit, RX_FRAME_t *frame, RX_BUFFER_t *buf,
       uint8_t scramble);
130 uint8_t decode_frame(RX_FRAME_t *rx_frame, RX_BUFFER_t *rx_buf)
       ;
131 uint8_t verify_frame(RX_BUFFER_t *rx_buf);
132
133 #endif
```